Clube Delphi +PHP

Ano 8 · Edição 100 · R$11,90

DevMedia group

Especial

100

Edição Especial

Boa Idéia

Crie um componente para realizar pesquisas prontas no banco de dados

Controle de Hardware com o Delphi
Aprenda a controlar um LCD

Expert
Crie formulários de cadastro automaticamente

Banco de Dados
Extraindo os metadados do BD via código

Easy Delphi
Paleta Servers – Integração com o Word e Excel

PHP
Reaproveitamento de código com a criação de classes

Na Web

Confira no portal ClubeDelphi PLUS 3 vídeo-aulas:

- Delphi for PHP e Firebird
- ASP.NET 2.0 e Stored Procedures
- Incorporando uma DLL a um executável

XML

ISSN 1517990-7

©2007 Microsoft Corporation. Todos os direitos reservados. Microsoft, Visual Studio, o logo do Visual Studio e "Seu potencial. Nossa inspiração". são marcas comerciais, registradas ou não, da Microsoft Corporation nos Estados Unidos e/ou em outros países. Os nomes das companhias ou produtos reais aqui mencionados podem ser marcas comerciais de seus respectivos proprietários.

Seu potencial. Nossa inspiração.™
Microsoft®
ENFRENTE OS DESAFIOS
Desafio: terminar projetos com qualidade dentro dos prazos. Estratégia: mais controle e previsibilidade no processo de desenvolvimento com o Visual Studio® Team System. Veja dicas e ferramentas em
enfrenteosdesafios.com.br
Microsoft®
Visual Studio®

Ano 8 - 100ª Edição - 2008 - ISSN 1517990-7 Impresso no Brasil

Corpo Editorial

Editor Geral
Guinther Pauli
guinther@devmedia.com.br

Editor Técnico
Adriano Santos
adrianosantos@devmedia.com.br

Equipe Editorial
Fabricio Desbessel, Maikel Scheid, Paulo Quicoli, Luciano Pimenta

Editor de Arte
Vinicius O. Andrade
viniciusoandrade@gmail.com

Diagramação e Capa
Vinicius O. Andrade
viniciusoandrade@gmail.com

Distribuição
Fernando Chinaglia Dist. S/A
Rua Teodoro da Silva, 907
Grajaú - RJ - 206563-900

Atendimento ao Leitor

A DevMedia conta com um departamento exclusivo para o atendimento ao leitor. Se você tiver algum problema no recebimento do seu exemplar ou precisar de algum esclarecimento sobre assinaturas, exemplares anteriores, endereço de bancas de jornal, entre outros, entre em contato com:

Carmelita Mulin
www.devmedia.com.br/central/default.asp
(21) 3382-5025

Kaline Dolabella
Gerente de Marketing e Atendimento
kalined@terra.com.br
(21) 3382-5025

Publicidade

Para informações sobre veiculação de anúncio na revista ou no site entre em contato com:

Kaline Dolabella
publicidade@devmedia.com.br

Fale com o Editor

É muito importante para a equipe saber o que você está achando da revista: que tipo de artigo você gostaria de ler, que artigo você mais gostou e qual artigo você menos gostou. Fique a vontade para entrar em contato com os editores e dar a sua sugestão!
Se você estiver interessado em publicar um artigo na revista ou no site ClubeDelphi, entre em contato com os editores, informando o título e mini-resumo do tema que você gostaria de publicar:

Guinther Pauli - Editor da Revista
guinther@devmedia.com.br

EDITORIAL

É com muita emoção que escrevo este editorial. Nós por aqui estamos muito orgulhosos e felizes, e claro, comemorando muito. É um sonho para qualquer editor estar à frente da maior e melhor revista sobre Delphi do país, chegando a nada mais nada menos que 100 edições! Foram muitos anos do melhor e mais puro Delphi, muito código, muitas aplicações, mini-cursos, vídeo-aulas e artigos. E o motivo desse sucesso todo é você amigo leitor, que sempre esteve ao nosso lado. Confira nessa edição uma retrospectiva que preparei para você relembrar os melhores momentos do Delphi e da ClubeDelphi.

Uma edição especial merece artigos muito especiais. Nossa matéria de capa da centésima edição vai para um dos temas mais populares e utilizados atualmente. A XML é a linguagem base para muitas tecnologias: Web Services, SOAP, intercâmbio de dados, integração entre diferentes plataformas, arquivos de configuração, DataSnap, ClientDataSet, ADO.NET etc. É fundamental hoje em dia que qualquer desenvolvedor saiba ler e gravar arquivos XML, pois mais cedo ou mais tarde uma integração com outro aplicativo/plataforma será necessária. Aprenda como fazer transformações XML com XML Mapper, utilizar o XMLDocument, usar XSLT e XML Data Bindings. Tudo isso no artigo do Manoel Edésio.

O Gustavo Chaurais na sessão Expert mostra como criar um mecanismo automático para geração de formulários de cadastro. Sem dúvida uma boa idéia e boa prática que pode poupar muito tempo de desenvolvimento. O Maikel mostra como trabalhar com metadados do BD, essencial para você explorar via código da aplicação a estrutura do seu banco. Quem sabe você não cria seu próprio IBExpert?! Outra aplicação interessante para a técnica seria criar atualizadores automáticos da base de dados quando novos updates do software forem lançados, comparando para isso os metadados.

Reutilização de código e funcionalidade é muito importante em aplicações Delphi, que usem ou não conceitos de OO. Um dos principais fundamentos do Delphi, desde a sua versão 1, foi o conceito de componentização. O Luciano mostra em seu artigo como criar um componente que encapsula formulários de pesquisa, de forma reutilizável. Além disso, o Thiago traz uma interessantíssimo artigo que mostra como controlar hardware com Delphi, nesse caso um LCD.

Quer Web? Temos ASP.NET 2.0 e PHP para você. Aprenda a usar Themes e Skins para dar diferentes formas para seu Web Site, da mesma forma que você troca um tema do Windows. Na sessão PHP, confira as novidades da versão 2.0 do Delphi for PHP, e também um artigo que mostra como trabalhar com classes no PHP. Para os iniciantes, na sessão Easy Delphi, veja como usar os componentes da paleta Servers para integração com o Word e Excel.

Desejo a todos muito sucesso com o Delphi, um muito obrigado e que venham mais 100 edições!

Guinther Pauli
guinther@devmedia.com.br
Microsoft Certified: MCP, MCAD, MCSD.NET, MCTS, MCPD
Delphi Certified: 6, 7, 2005, 2006, Web, Kylix

A revista ClubeDelphi é parte integrante da assinatura ClubeDelphi PLUS.
Para mais informações sobre o pacote PLUS, acesse:
http://www.devmedia.com.br/clubedelphi/portal.asp

Sumário

Olá, eu sou o **DevMan**! Desta página em diante, eu estarei lhe ajudando a compreender com ainda mais facilidade o conteúdo desta edição. Será um prazer contar com sua companhia! Confira abaixo o que teremos nesta revista:

Delphi

Delphi Core

10 – XML no RAD Studio
Aprendendo o máximo de XML
[*Manoel Edésio*]

Expert

20 – Criação Automatizada de Formulários
Agilize o processo de criação de cadastros usando técnicas avançadas de automatização de tarefas
[*Gustavo Chaurais*]

Banco de Dados

32 – Metadados
Veja como recuperar a estrutura do seu banco de dados através da sua aplicação
[*Maikel Scheid*]

Banco de Dados

38 – Criando um componente de pesquisa
Reutilize as telas de pesquisa em sua aplicação
[*Maikel Scheid*]

Boa Idéia

44 – Controle de Hardware com Delphi
Usando um LCD no Delphi
[*Thiago Batista Limeira*]

Easy Delphi

Easy Delphi

54 – Introdução à paleta Servers
Integre sua aplicação com Word e Excel
[*Maikel Scheid*]

PHP

PHP

62 – Trabalhando com classes no PHP
Veja como implementar e utilizar na prática classes em Web Sites PHP
[*Adriano Santos*]

Você percebeu os nomes ao lado de cada matéria? Eles indicam o que você vai encontrar no artigo – dessa forma, você também pode ter uma idéia geral do que vai encontrar nesta edição como um todo! Os editores trabalham sempre no sentido de fechar a revista seguindo esta definição, para oferecer a você o melhor conteúdo didático!

Confira abaixo a lista com a definição dos tipos de artigo encontrados nesta edição:

[Mão na Massa] Artigos que focam na resolução de problemas - e não na tecnologia. A tecnologia empregada é apenas conseqüência. O artigo parte do pressuposto que o leitor já conhece a tecnologia, mas tem dificuldade de implementá-la em uma aplicação cotidiana. Por exemplo, um artigo que ao invés de "debulhar" todos os métodos e propriedades de um "Recordset", mostre como usar o Recordset de forma inteligente. para criar um carrinho de compras.

[Mini-Curso] Artigo que faz parte de um mini-curso.

[PHP] Artigo com foco no PHP puro.

[Boas Práticas] A Um dos objetivos da revista é levar para o leitor não somente as melhores técnicas, mas também as melhores práticas – Esse tipo de artigo foca em técnicas que poderão aumentar a qualidade do desenvolvimento de software.

[Web] Artigos sobre ou que envolvam técnicas de desenvolvimento para WEB.

[Expert] Artigo com foco no leitor avançado.

[Easy Delphi] Com foco no desenvolvedor iniciante.

[Novidades] Artigos sobre tecnologias de ponta, que ainda não fazem parte do dia a dia do desenvolvedor mas que prometem ser a próxima febre.

[Boa Idéia] A Esse tipo de artigo é quase um artigo "Mão na Massa" - a diferença é que o exemplo do artigo é tão criativo que é considerado uma 'boa idéia', dentro do contexto de desenvolvimento de software.

[Banco de Dados] Artigo sobre o universo de persistência de dados.

Assinatura
ClubeDelphi PLUS
Mais conteúdo Delphi por menos

Informativo ClubeDelphi

Portal ClubeDelphi

www.clubedelphi.net/portal

+600 vídeo aulas e 7 cursos online

Brinde na web desta edição

3 Vídeos

ASP.NET 2.0 e Stored Procedures
http://www.devmedia.com.br/articles/viewcomp.asp?comp=9982

Delphi for PHP e Firebird
http://www.devmedia.com.br/articles/viewcomp.asp?comp=9537

Incorporando uma DLL a um executável
http://www.devmedia.com.br/articles/viewcomp.asp?comp=8393

Para acessar os vídeos utilize os seguintes dados: Login: **DVM.CD** Senha: **PEG*VDC**

Gostou das vídeo aulas? O portal *www.devmedia.com.br* possui mais de **2 mil vídeo aulas** e **dezenas de cursos online** sobre desenvolvimento de software! Agora você pode comprar as vídeo aulas que preferir e fazer sua própria combinação de vídeos! Saiba mais em *www.devmedia.com.br/creditos*

Edições Anteriores da ClubeDelphi

Você pode comprar todas as edições anteriores da ClubeDelphi através do site DevMedia!

Para isso basta acessar *https://seguro.devmedia.com.br/edicoes_anteriores.asp*

Dê seu feedback sobre esta edição!

A ClubeDelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre esta edição, artigo por artigo, através do link: *www.devmedia.com.br/clubedelphi/feedback*

Para votar, você vai precisar do código e senha de banca desta edição, que é:

Login: **DVM.CD** Senha: **PEG*VDC**

Hospede com quem fala sua língua!

Local host

Hospedagem Profissional
Revenda
Parcerias

Ativação imediata!

Faça sua assinatura, tenha 30 dias grátis

e seu primeiro boleto para 60 dias após assinatura em qualquer plano

UNIX I

a partir R$ 28,00

- 1 GB Espaço em disco
- 60 GB Transferência
- 30 Contas de e-mail
- 10 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

UNIX II

a partir R$ 58,00

- 3 GB Espaço em disco
- 120 GB Transferência
- 60 Contas de e-mail
- 15 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

UNIX III

a partir R$ 98,00

- 6 GB Espaço em disco
- 300 GB Transferência
- 90 Contas de e-mail
- 20 Domínios
- Painel de Controle em Português
- Atendimento exclusivo

Contratar

R$ 5,90 — Mini html

Tráfego 1 GB
Espaço 100 MB
3 C.Ppostal, Espaço 3G
Domínio 3

R$ 15,90 — html

Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado

R$ 15,90 — ASP

Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado
MySql 50m

R$ 20,90 — PHP

Tráfego 25 GB
Espaço 500 MB
20 C.Ppostal, Espaço 20G
Domínio Ilimitado
MySql 50m

Formato exclusivo de trabalho para empresas de desenvolvimento de sites, websites, portais, serviços e negócios que necessitam de grandes tráfegos, monitoramento e backup.

www.lhost.com.br

assinar@lhost.com.br

Retrospectiva ClubeDelphi

100 Edições de puro sucesso

Guinther Pauli

é Bacharel em Sistemas de Informação, Microsoft Certified: MCP, MCAD, MCSD.NET, MCTS, MCPD, Borland Certified: Delphi 6, 7, 2005, 2006, Web e Kylix. Editor Geral das Revistas .net Magazine e ClubeDelphi. Trabalha com a ClubeDelphi desde a edição número 19 e com Delphi desde quando era um protótipo chamado Borland Pascal for Windows.

Me sinto completamente à vontade para falar de Delphi e ClubeDelphi, nesta centésima edição. Pudera, acompanho ambos desde os seus primórdios e presenciei todas as fases de sucesso dessa eficiente IDE/Linguagem e dessa fabulosa Revista. Nessa retrospectiva, veja um pouco de todo o nosso longo e árduo trabalho trazendo sempre o melhor do Delphi a cada mês.

A ClubeDelphi número 1 foi lançada em 2000 e já na primeira edição mostrava a sua marca registrada: cursos passo a passo estilo tutorial. O famoso "Controle de Biblioteca" foi a primeira aplicação a ser desenvolvida na ClubeDelphi. Usava como banco de dados o poderoso (na época) Paradox 7! Aliado ao BDE e Delphi 6, controlava livros, empréstimos e outras funcionalidades de uma biblioteca. Também já constava a tradicional coluna Componente do Mês, trazendo a suíte MMTools. E que tal "DLLs? Devo mesmo usar?". Bons tempos em que as DLLs faziam sucesso (antes do DLL Hell). As edições seguintes continuaram seguindo as tendências do mercado, muito Paradox, SQL (quem não lembra dos famosos utilitários BDE Administrator, Database Desktop, SQL Explorer, DataPump etc.). E para não deixar de fora, o mais famoso e utilizado gerador de relatórios com o Delphi, o QuickReport, abordado desde as primeiras edições.

Entrando no mundo client/Server, as edições seguintes abordaram o uso do Interbase, banco de dados mais utilizado pelos desenvolvedores Delphi até o surgimento do Firebird. Componentes da paleta IBX, que vieram no Delphi 5, faziam sucesso por não dependerem do BDE e fazerem acesso direto ao Interbase.

Já surgiam os primeiros sinais do mundo multicamadas, sendo que o

ClientDataSet foi abordado em detalhes. Na edição 19, abordamos o uso do COM/DCOM para construções de aplicações distribuídas com MIDAS (*Multitier Distributed Application Services*), hoje conhecido como DataSnap. Bons tempos! O COM+ entrou em ação logo a seguir, mostrando todo o seu poder na programação de objetos distribuídos.

A Web, claro, não poderíamos deixar de falar dela. WebBroker, do Delphi 5, com seus Producers e WebModules na construção de CGI`s e ISAPI`s foi sucesso na época. O WebSnap, com seus Adapters trouxe uma revolução no Delphi 6 para o desenvolvimento para Internet. Sem falar do Internet Express e XMLBroker do Delphi 5 (tivemos uma série sobre eles), que permitiam a integração de clientes browser com servidores de aplicação MIDAS fazendo cache local em XML com JavaScript, algo que até hoje não vi em outra tecnologia.

Não poderíamos deixar passar em branco o surgimento do Kylix, abordado na edição 20, o famoso "Delphi para Linux". Com a biblioteca CLX, juntamente com o Delphi 6, era possível construir aplicações cross-platform sendo que um mesmo código fonte poderia ser compilado tanto no Linux quanto no Windows. E o Kylix foi um grande produto, um dos melhores IDEs para Linux já construídos.

Ambos Kylix e Delphi 6 trouxeram um novo e poderoso engine de acesso a dados, o dbExpress, tratado em detalhes na edição 22. O dbExpress fazia acesso nativo aos principais bancos do mercado, incluindo Oracle, DB2, SQL Server, Interbase etc. Junto com o dbExpress, muitos artigos cobriram os recursos do ClientDataSet, personagem indispensável quando se trabalha com essa tecnologia (e um dos melhores componentes do Delphi que já vi).

O DataCar, na edição 29, foi um dos mais populares sistemas já desenvolvidos na ClubeDelphi (seus fontes estão disponíveis para download até hoje). Utilizava boas práticas de desenvolvimento client/server com Firebird e rodava com Delphi 7 e Kylix 3. A essa altura, o Kylix 3 já suportava Web Services e a linguagem C++.

Uma edição especial (33) sobre o Delphi 7 foi lançada, onde todos os recursos foram abordados em detalhes. Em especial, para quem lembra, o Delphi 7 trouxe o framework para desenvolvimento Web chamado IntraWeb. Também foi incorporado ao produto o Rave Reports, ModelMaker e Bold (para construção de aplicações MDA e OO). Também foi nessa versão que foi incluído o primeiro preview do compilador do Delphi para .NET, e a ClubeDelphi trouxe desde aquela época artigos sobre o .NET Framework, quando nem IDE tínhamos.

Na edição 48 novamente uma cobertura completa sobre o novo Delphi, agora o Delphi 8 for .NET. Além de abordar o novo IDE, totalmente remodelado, novas tecnologias foram discutidas nessa edição e seguintes, entre elas: ASP.NET Web Forms, ASP.NET Web Services, Windows Forms, ADO.NET, VCL.NET, ECO e muito mais. A ClubeDelphi foi pioneira não só no mundo Win32, mas no universo .NET também.

A edição 57 (**Figura 1**) foi outra edição histórica (além de ter a melhor arte de capa que já vi): em uma única edição conseguimos compilar só artigos sobre o novíssimo Delphi 2005, que uniu em um único IDE suporte para Delphi Win32, Delphi for .NET e C# Builder. Tratamos todos os novos recursos do IDE, como melhorias, .NET Remoting e BDP, ECO II, StarTeam, Refactoring, DUnit, ASP.NET e migração. Nesse período a ClubeDelphi também esteve na BorCon americana para acompanhar de perto o lançamento do produto e trazer as novidades em primeira mão.

Na edição 60 um super artigo com nada mais nada menos que 60 dicas de Delphi! Também um belo artigo de como configurar e trabalhar com o StarTeam. Nessa edição também estava em andamento um curso de ASP.NET, esse que considero o melhor framework para desenvolvimento Web. Também abordamos até a criação de games com o Delphi!

As edições seguintes tiveram muitos mini-cursos: SysCash, OOPark, SysPague, Locadora on-line, Loja Virtual com

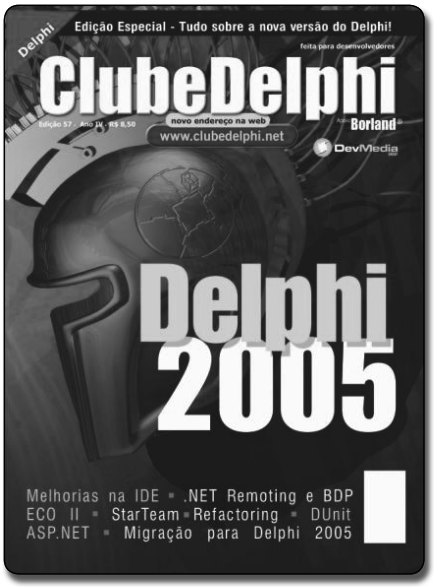


Figura 1. Capa da edição 57

UML e ECO, DataCar OO (mais recentemente) entre outros. Muitos artigos sobre boas práticas de desenvolvimento, programação multicamadas com DataSnap, desenvolvimento para Web com ASP.NET, Web Services / SOAP, muito Win32, Client/Server com Firebird, SQL Server e Oracle, dbExpress, desenvolvimento Mobile para Palm e PocketPC, Testes, Engenharia de Software, Projeto/Análise, Mão na Massa, Banco de Dados e muito mais.

Nas últimas edições fizemos melhorias editoriais, aumentamos o número de páginas, incluímos as sessões Easy Delphi para o público iniciante e também a sessão PHP. Nosso portal hoje é sem dúvida o maior do Brasil para desenvolvedores com centenas de vídeo-aulas.

E aqui estamos nós, edição número 100 da maior e melhor revista sobre Delphi do país! Nosso compromisso? Continuar trazendo para você leitor o melhor do Delphi a cada mês, independente do tipo de aplicativo que desenvolva: Web, client/server, multicamadas, com a certeza de estar usando a melhor linguagem de todos os tempos. E o motivo disso tudo? Você caro amigo leitor, que sempre esteve ao nosso ao lado. Um sincero muito obrigado!

Um grande abraço a todos e que venham mais 100 edições! ●

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

XML

Aprendendo o máximo de XML

Resumo DevMan

Sem dúvida nenhuma a utilização de arquivos texto, os famosos TXT, têm sido utilizados há anos por diversas empresas para fazer a integração entre diferentes aplicações. Um bom exemplo disso são os EDI's (Eletronic Data Interchange) usados por bancos. Os EDI's também são chamados de CNAB's e o seu principal objetivo é fazer a emissão de boletos e cobranças bancárias. A software house se dispõem a desenvolver a exportação de dados de seu sistema para cobrança no formato texto especificado pela instituição financeira.

Partindo desse conceito, outras empresas passaram a utilizar desse tipo de recurso como empresas de contabilidade, por exemplo. Porém, a manipulação de arquivos texto é complexa, demorada e bastante propícia a erros.

Neste artigo veremos como fazer a utilização de arquivos XML que possuem formato universal e têm maior controle da integridade de dados.

Nesse artigo veremos

- Introdução à XML;
- Manipulação de arquivos XML com XMLDocument;
- XML Transformer;
- XML com Interfaces.

Qual a finalidade

- Entender o que é um arquivo XML, como manipular via XMLDocument, trabalhando com XML no ClientDataSet, trabalhando com XML e interfaces;

Quais situações utilizam esses recursos?

- Troca de arquivos entre aplicações, gravação de dados locais para tabelas temporárias etc.;

Manoel Edesio B da Silva

manoel.edesio@hkconsultoria.com.br

detém as certificações em Borland C++ Builder, Borland Delphi for Win32 e Borland Delphi for .NET. Palestrou para a Borland em Delphi Meetings em alguns estados brasileiros, palestrante na Borland Conference de 2004, 2005, 2006 e 2007. Atualmente é diretor da HK Consultoria, empresa especializada em consultoria, treinamentos customizados e desenvolvimento de software em produtos Borland. Além dos quesitos anteriores a HK atua com projetos especializados em mobilidade com utilização de celulares e PDA'S

XML (*Extensible Markup Language*) é um formato de arquivo texto que hoje é praticamente o mais utilizado e recomendado para troca de informações entre aplicativos. Este seria um resumo de forma simples sobre XML, mas não podemos falar desse maravilhoso recurso sem fazer um paralelo com o bom e velho amigo arquivo *TXT*.

Todos nós desenvolvedores já precisamos alguma vez em algum projeto ou em alguma situação específica fazer a leitura de um arquivo *TXT*. Senão foi a leitura foi a geração de um arquivo para disponibilizar dados para outras

aplicações, integrações com programas da CEF (*Caixa Econômica Federal*), por exemplo, ou para guardar informações próprias para nosso uso. Sabemos que ao manipular um arquivo desse tipo temos diversas dificuldades de manipulação, pois para se fazer, por exemplo, a leitura do mesmo, precisamos definir sempre uma regra. Ou definimos um caractere delimitador para determinar quando termina um dado e quando inicia outro, ou então trabalhamos com posicionamento dos caracteres por ex: da posição 1 até a 5 é o código, da posição 6 até a 40 é o nome e por aí vai.

Sempre que temos esse tipo de modelo o nosso código sempre fica fixo para o tratamento de um determinado arquivo X, se o proprietário desse arquivo mudar o formato lá vai o programador novamente refazer o seu código para ajustar aquele padrão. Na verdade o que sempre acontece é que só detectamos a mudança do mesmo em produção o que gera diversos constrangimentos ao nosso cliente/usuário final.

Além desses detalhes aqui mencionados temos muitas dificuldades para leitura e gravação, imagine ter que ler um arquivo TXT com 1000 linhas que seja, e localizar um registro lá dentro que tem o nome de Juliana e ainda ter que trocar um item na linha que representa o salário para um novo valor. Dá para ter noção do trabalho? E o pior, quem é que garante a integridade dessa informação. Eu não garanto e Você?

Através de um arquivo XML temos inúmeras vantagens que não temos em outros formatos, o primeiro ponto é que é um modelo de arquivo que segue uma padronização, além disso, é multi-plataforma e independente de linguagem. Um arquivo XML é apenas um arquivo de texto que é delimitado por TAGS criados pelo próprio usuário, o documento em si não faz coisa alguma só armazena o conteúdo, tudo é pura informação.

Podemos criar um arquivo XML em qualquer editor de textos, depois de criado o arquivo, escrevemos código para interpretá-lo. Porém, temos diversas ferramentas para criar um arquivo XML, interpretá-lo, componentes para fazer leitura, escrita, exibição e etc.

Podemos citar como principais vantagens de se trabalhar com um arquivo XML:

- Inteligente (tags personalizadas);
- Manutenção facilitada;
- Simplicidade;
- Portabilidade.

Em determinado ponto podemos comparar um arquivo XML com um arquivo HTML e podemos citar as principais diferenças já que o XML também faz uso de tags para deixarmos as definições de forma bem clara.

HTML

- Concentra-se no uso de tags HTML para formatar e exibir informações;
- Usa Tags pré-definidas. As tags especificam como os dados entre elas irão aparecer no browser;
- Em HTML os dados estão gravados dentro do próprio texto do arquivo.

XML

- Descreve informações;
- Usa tags simplesmente para delimitar grupo de dados;
- O usuário tem a habilidade para criar as suas próprias tags;
- Os dados em um arquivo XML podem estar em um arquivo separado, em qualquer lugar do disco;
- Dados em um XML podem ser manipulados através de rotinas, *procedures* que interpretem o documento *similar a SQL para database*.

Entendendo o documento XML

Na **Listagem 1** temos um modelo de um arquivo XML bem simples para começarmos a entender um pouco sobre a estrutura do documento. Na primeira linha do documento temos a declaração XML, definindo a versão que está sendo usada. Segunda linha, elemento *root* (descreve o tipo do documento), posteriormente temos os elementos *Childs* do *root* (equivalentes aos campos). Para cada tag aberta, deve existir uma tag respectiva de fechamento. Por ex.:

- * Parágrafo. Isto é válido em html;*
- * sintaxe válida para XML ;*

Nota do DevMan

A troca de informações no formato TXT entre empresa também é por EDI. EDI é a sigla de Electronic Data Interchange, que em português significa Troca Eletrônica de Dados. É a troca de documentos via sistemas de teleinformática entre duas ou mais organizações de forma padronizada. Tem como um dos principais objetivos, ao substituir o fluxo de papéis entre elas, agilizar e reduzir os custos dos processos mercantis.

Tradicionalmente, no âmbito das relações comerciais, muitas das aplicações informatizadas de uma empresa geram documentos em papel que, após o trâmite burocrático interno, são encaminhados a outra empresa. Nesta outra empresa, tais documentos passam inicialmente por uma averiguação e então são digitados e introduzidos nos seus sistemas informatizados.

Existem informações nos dando conta de que 70% das entradas de dados do computador de uma empresa são originados pelas saídas de computadores de outras empresas e de que 25% do custo das transações derivam destas entradas.

O uso do EDI tem como proposta automatizar estas transações comerciais e eliminar este grande número de papéis que são trocados entre empresas. Para que se consiga isto, é necessário conectar diretamente os computadores das empresas que transacionam entre si.

Nota: A primeira tag que identifica que se trata de um documento XML (<xml>) não é fechada no final. Isto porque esta linha não faz parte do documento em si, não é um elemento XML, por isso não tem seu respectivo fechamento.

Todas as tags são *case sensitive*, diferente da linguagem HTML:

- *<P> Isto é incorreto ;*
- * Isto é correto ;*

Elementos combinados (*nested*) devem respeitar uma ordem para abertura e respectivos fechamentos:

- * Isto é incorreto *
- * Isto é correto *

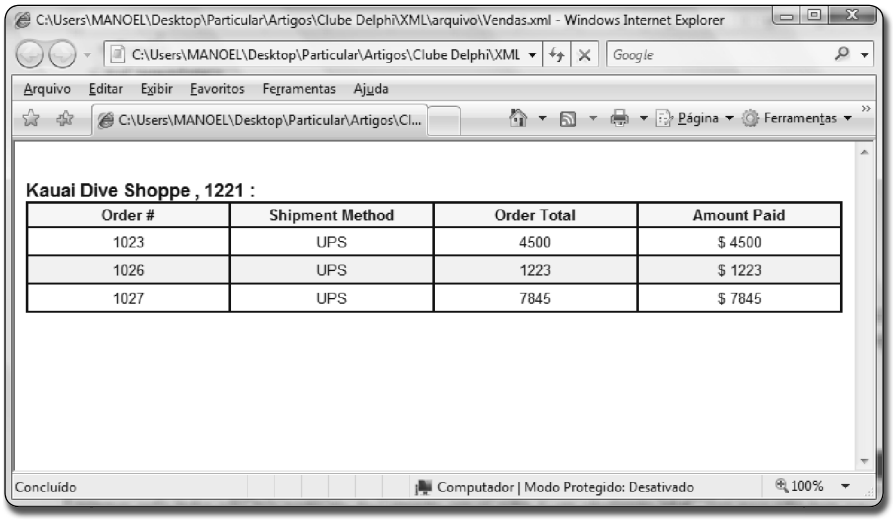


Figura 1. Arquivo XML sendo formatado pelo XSL

Listagem 1. Modelo de Arquivo XML

```
<?xml version="1.0" encoding="UTF-8" ?>
<Cadastro>
 <Empregado codigo="1" Dep="Tecnico">
  <Nome>Juliana</Nome>
  <Cargo>Gerente de Projetos</Cargo>
  <Sexo>F</Sexo>
  <Salario>4500</Salario>
</Empregado>
<Empregado codigo="2" Dep="Executivo">
  <Nome>Manoel</Nome>
  <Cargo>Diretor</Cargo>
  <Sexo>M</Sexo>
  <Salario>5000</Salario>
</Empregado>
<Empregado codigo="3" Dep="Comercial">
  <Nome>Beatriz</Nome>
  <Cargo>Analista de Processos</Cargo>
  <Sexo>F</Sexo>
  <Salario>4000</Salario>
</Empregado>
</Cadastro>
```

Listagem 2. Arquivo XML com formatação de um XSL

```
<?xml version="1.0" ?>
<?xml:stylesheet type="text/xsl" href="VendasXSL.xsl"?>
<Document>
<Customer>
 <CUSTNO>1221</CUSTNO>
  <COMPANY>Kauai Dive Shoppe</COMPANY>
<Orders>
<OrderRecord>
  <ORDERNO>1023</ORDERNO>
  <CUSTNO>1221</CUSTNO>
  <SHIPVIA>UPS</SHIPVIA>
  <ITEMSTOTAL>4500</ITEMSTOTAL>
  <AMOUNTPAID>4500</AMOUNTPAID>
</OrderRecord>
<OrderRecord>
  <ORDERNO>1026</ORDERNO>
  <CUSTNO>1221</CUSTNO>
  <SHIPVIA>UPS</SHIPVIA>
  <ITEMSTOTAL>1223</ITEMSTOTAL>
  <AMOUNTPAID>1223</AMOUNTPAID>
</OrderRecord>
<OrderRecord>
  <ORDERNO>1027</ORDERNO>
  <CUSTNO>1221</CUSTNO>
  <SHIPVIA>UPS</SHIPVIA>
  <ITEMSTOTAL>7845</ITEMSTOTAL>
  <AMOUNTPAID>7845</AMOUNTPAID>
</OrderRecord>
</Orders>
</Customer>
</Document>
```

A primeira tag do documento XML indica o elemento *root*, que deve ser devidamente fechada ao final. A partir dela, são criados outros pares de tags, que se constituem nos elementos do documento. Os elementos podem possuir sub-elementos (*children*):

```
<root>
    <child>
        <subchild></subchild>
    </child>
</root>
```

Os elementos podem possuir atributos. Estes valores devem estar entre aspas, como vemos a seguir:

```
<?xml version = "1.0"?>
<memo>
        <para>Juliana</para>
        <envio>sent = "17/10/2007"</envio>
        <mens>HK Consultoria -
Treinamentos!</mens>
</memo>
```

Espaços em branco não são desprezados, como acontece com HTML:

```
<mens>Teste de envio!</mens>
```

> **Nota:** A mensagem *"Teste de envio!"* que vimos anteriormente, será exibida da como foi que foi escrita, ao passo que, com HTML, os espaços em branco seriam truncados.

Nomeando elementos XML

- Os nomes podem conter letras, números e outros caracteres;
- Não podem iniciar com números ou caracteres especiais (pontuação, etc.);
- Não podem iniciar com "xml", "XML" ou "Xml";
- Não podem conter espaços.

Atributos XML

- Devem vir entre aspas duplas ou simples. As duplas são mais comuns. As simples são mais usadas para combinar (*nested*). No caso do próprio atributo já possuir aspas em seu conteúdo:

```
<person name='Johnny "Quest" Jones'>
```

XSL (Extensible Stylesheet Language)

- Similar ao que os arquivos CSS fazem com o HTML, o XSL é um tipo de arquivo que tem como propósito formatar um arquivo XML,

porém tem uma enorme vantagem de sempre ser um arquivo externo, além de se formatar a exibição do arquivo XML, por exemplo, no *browser* é permitido ainda ter diversas instruções típicas de linguagens de programação para vários propósitos diferentes.

Formatando um XML com XSL

Temos na **Listagem 2** um Arquivo XML que possui elementos *childs*. Se o mesmo fosse aberto no *browser* teria a aparência nada muito diferente do que na listagem a seguir, porém adicionando a tag que existe na segunda linha estamos vinculando ao XML um arquivo XSL de formatação que se encontra na **Listagem 3**, com isso teremos o resultado surpreendente no *browser* apresentado na **Figura 1**.

Explicando rapidamente, o arquivo XSL relacionado ao XML que chama-se *VendasXSL.xsl* tem dois laços *for-each* que percorre tanto todos os clientes quanto todos os pedidos, o mesmo formata cada campo com atributos, o mesmo faz o famoso *cor sim/cor não* que pode ser notado na linha **<xsl:when test="position() mod 2 = 0">** e como resultado final temos a fantástica exibição.

XML no RAD Studio

Depois de toda a explicação básica agora vamos começar a trabalhar com o arquivo XML no RAD Studio, iniciando através de um exemplo. Uma coisa comum ao trabalhar com arquivos XML é fazer a leitura do arquivo e exibir o mesmo em um componente *TreeView*. Vamos no RAD Studio acessar o menu *File>New>VCL Forms Application Delphi for Win32*. Após criada aplicação salve a mesma através do menu *File>Save all* e salve o projeto com o nome de *XmlParsing.dpr* e a *Unit* com o nome de *unitFrmParsing.pas*. Em seguida adicione um componente *OpenDialog*, um *TreeView*, um *XmlDocument*, um *Label* e um *Button*. Mude o nome do *label* para *lblArquivo* e o nome do *OpenDialog* para *OpnDlgAbrir*. Acesse o *OpenDialog* novamente e na propriedade *Filter* adicione um filtro cujo

Listagem 3. Arquivo XSL para formatação do XML

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/
XSL/Transform">
  <xsl:template match="/">
    <html>
      <body>
        <xsl:for-each select="Document/Customer">
          <br></br>
          <font face="Arial" size="3">
            <b>
              <!--CUSTOMER-->
              <xsl:value-of select="COMPANY"/>
              ,
              <!--ID-->
              <xsl:value-of select="CUSTNO"/>
              :
            </b>
          </font>
          <br></br>
          <!-- Table of Customer's Orders -->
          <table width="100%" border="1" cellspacing="0"
             cellpadding="2" bordercolor="#000000">
            <!--HEADER ROW -->
            <tr bgcolor="#FFFFCC" bordercolor="#000000">
              <td width="25%">
                <div align="center">
                  <b>
                    <font face="Arial" size="2">Order #</font>
                  </b>
                </div>
              </td>
              <td width="25%">
                <div align="center">
                  <b>
                    <font face="Arial" size="2">Shipment Method</font>
                  </b>
                </div>
              </td>
              <td width="25%">
                <div align="center">
                  <b>
                    <font face="Arial" size="2">Order Total</font>
                  </b>
                </div>
              </td>
              <td width="25%">
                <div align="center">
                  <b>
                    <font face="Arial" size="2">Amount Paid</font>
                  </b>
                </div>
              </td>
            </tr>
            <xsl:for-each select="Orders/OrderRecord">
              <tr>
              <xsl:choose>
                <xsl:when test="position() mod 2 = 0">
                  <xsl:attribute name="bgcolor">#F4F4F4</xsl:attribute>
                  <xsl:attribute name="bordercolor">#000000</xsl:attribute>
                </xsl:when>
                <xsl:otherwise>
                  <xsl:attribute name="bordercolor">#000000</xsl:attribute>
                </xsl:otherwise>
              </xsl:choose>
                <td width="25%">
                  <div align="center">
                    <font face="Arial" size="2">
                      <a>
                        <xsl:value-of select="ORDERNO"/>
                      </a>
                    </font>
                  </div>
                </td>
                <td width="25%">
                  <div align="center">
                    <font face="Arial" size="2">
                      <xsl:value-of select="SHIPVIA"/>
                    </font>
                  </div>
                </td>
                <td width="25%">
                  <div align="center">
                    <font face="Arial" size="2">
                      <xsl:value-of select="ITEMSTOTAL"/>
                    </font>
                  </div>
                </td>
                <xsl:choose>
                  <xsl:when test="AMOUNTPAID &lt; ITEMSTOTAL">
                    <td width="25%" bgcolor="RED">
                      <div align="center">
                        <font face="Arial" size="2">
                          $
                          <xsl:value-of select="AMOUNTPAID"/>
                        </font>
                      </div>
                    </td>
                  </xsl:when>
                  <xsl:otherwise>
              <td width="25%">
                <div align="center">
                  <font face="Arial" size="2">
                    $
                    <xsl:value-of select="AMOUNTPAID"/>
                  </font>
                </div>
                      </td>
                    </xsl:otherwise>
                  </xsl:choose>
                </tr>
              </xsl:for-each>
            </table>
          </xsl:for-each>
        </body>
      </html>
    </xsl:template>
</xsl:stylesheet>
```

o *Filter Name* seja *Arquivos XML* e o Filter seja **.xml*. Na propriedade *DefaultExt* configure para XML e na propriedade *Title* coloque *Abrir Arquivo XML*.

Feita a primeira configuração podemos conferir o resultado do design na **Figura 2.** Em seguida vamos acessar a nossa *Unit* e adicionar as duas declarações encontradas na seção *private* como a seguir:

```
private
  procedure AdicionaNo(NoXML : IXMLNode;
    Pai : TTreeNode);
  procedure LerArquivo(Nome : TFileName);
...
```

O método *adicionaNo* terá como propósito ler todo o arquivo XML em cada nível existente e montar nó a nó dentro do *TreeView*. Já o método *LerArquivo* terá o papel apenas de receber o arquivo XML de retorno do *OpenDialog* para abertura pelo *XMLDocument*. Após declarar os métodos vamos pressionar a combinação de teclas *Shift+Ctrl+C* para gerar as suas implementações, vamos começar implementando o primeiro método conforme **Listagem 4.**

No início do código após a declaração das variáveis iniciamos adicionando um primeiro nó ao *Treeview* que recebe como parâmetro o elemento Pai do tipo *TreeNode* que é o ponto de início do nosso trabalho e utilizamos a propriedade *NodeName* do *NoXml* para extrair o nome *root* que será trabalhado. Iniciamos um *for* para percorrer todos os atributos existentes naquele nó XML e passamos para variável Filho o conteúdo dela própria adicionado o *NodeName* que é o nome do nó XML, em seguida verificamos se o nó XML tem elementos filhos e caso tenha percorremos todos os elementos. Para cada elemento verificamos através de um *case* se o mesmo é do tipo *ntText* e se for, quer dizer que chegamos já no nível de conteúdo do nó XML, caso não seja desse tipo quer dizer que ainda existem elementos filhos no nó XML. Então pegamos o nó XML corrente e fazemos uma chamada recursiva para o próprio método para percorrer todos os itens que ele tem.

Em continuidade à **Listagem 4** temos o método *LerArquivo* que explicarei agora. Antes de iniciar o processamento do arquivo trocamos o cursor do mouse para *crHourGlass* que é a famosa "ampulheta", em seguida carregamos o arquivo XML dentro do nosso componente *XMLDocument* e setamos a propriedade *Active* para *True*, com isso já temos todo o conteúdo do arquivo XML carregado na memória do componente. Seguindo o código chamamos o método *Clear* do *TreeView* para limpar todos os itens que possam estar lá dentro e para efetuar o processamento de montagem do *TreeView* baseado no XML chamamos o método *AdicionaNo* passando como parâmetro o elemento *Root* do documento e como segundo parâmetro, que é o ponto de início do *TreeView* como *nil*. O *nil* servirá para informar ao nosso método que queremos iniciar a montagem do *TreeView* do ponto "zero".

Listagem 4. Implementação dos métodos AdicionaNo e LerArquivo

```
procedure TForm1.AdicionaNo(NoXML: IXMLNode; Pai : TTreeNode);
var
  SubNoXML : IXMLNode;
  Filho : TTreeNode;
  i : integer;
begin
  Filho := TreeView1.Items.AddChild(Pai, NoXML.NodeName);

  for i := 0 to NoXML.AttributeNodes.Count-1 do
    Filho.Text := Filho.Text + ' ' +
      NoXML.AttributeNodes.Get(i).NodeName + '=' +
      QuotedStr(NoXML.AttributeNodes.Get(i).Text);

  if NoXML.HasChildNodes then
    for i := 0 to NoXML.ChildNodes.Count - 1 do
    begin
      case NoXML.ChildNodes[i].NodeType of
        ntText : TreeView1.Items.AddChild(Filho, NoXML.Text);
      else
        SubNoXML := NoXML.ChildNodes[i];
        AdicionaNo(SubNoXML, Filho);
      end;
    end
end;

procedure TForm1.LerArquivo(Nome: TFileName);
begin
  Screen.Cursor := crHourGlass;
  try
    XMLDocument1.LoadFromFile(Nome);
    XMLDocument1.Active := true;
    TreeView1.Items.Clear;
    AdicionaNo(XMLDocument1.DocumentElement, nil);
  finally
    Screen.Cursor := crDefault;
  end;
end;
```

Listagem 5. Implementando o Click do Botão

```
procedure TForm1.spbtAbrirClick(Sender: TObject);
begin
  if OpnDlgAbrir.Execute then
  begin
    lblArquivo.Caption := ExtractFileName(OpnDlgAbrir.FileName);
    LerArquivo(OpnDlgAbrir.FileName);
  end;
end;
```

Figura 2. Design da tela do exemplo XMl Parsing

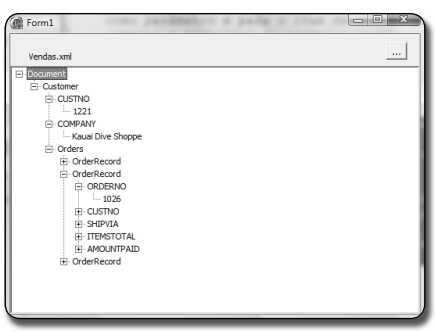


Figura 3. Aplicação do XML Parsing em execução

Em seqüência às implementações vamos dar um duplo clique no botão e implementar o código da **Listagem 5**. No início do código executamos o *OpenDialog* através do método *execute* e verificamos se o seu retorno é verdadeiro para dar continuidade ao processo. O resultado sendo *True* utilizamos o método *ExtractFileName* para extrair apenas o nome do arquivo que está na propriedade *FileName* do *OpenDialog* e passar o mesmo para a propriedade *Caption* do nosso *Label*. Para carregar o arquivo para o processamento executamos o método *LerArquivo* e passamos como parâmetro o conteúdo da propriedade *FileName* do *OpenDialog* para leitura do arquivo escolhido. Depois de implementado, poderemos conferir o resultado na **Figura 3.**

XML com ClientDataSet

Como sempre o nosso inseparável *ClientDataSet*, que considero o componente mais poderoso da VCL e com isso podemos esperar qualquer coisa dele, dá suporte à XML. Através desse componente facilmente podemos gerar dados que estão no seu *cache* para um arquivo XML do tipo *DataPacket*. Esses dados podem vir de um banco de dados, pode ser de uma tabela em memória etc. Estando no *cache* do *ClientDataSet* conseguimos salvar o conteúdo dele em um arquivo XML em disco ou até em um *Stream* de memória.

Para demonstrar essa funcionalidade criaremos uma nova aplicação RAD Studio. Acesse o menu *File>New >VCL Forms Application – Delphi for Win32*. Em seguida acesse o menu *File>Save All* e salve o projeto com o nome *XMLDataPacket.dpr* e a *Unit* com o nome de *unitFrmXmlDataPacket*. Em seguida, vamos adicionar ao formulário 3 componentes *Button, 1 DbGrid, 1 ClientDataSet* e um *DataSource*. Selecione o componente *DataSource* e aponte a propriedade *DataSet* para o *ClientDataSet*. Em seguida selecione o *DbGrid* e aponte a propriedade *DataSource* para o componente de mesmo nome. Depois configure o *Caption* dos botões conforme o *design* do formulário na **Figura 4**.

Em seguida vamos implementar os nossos três botões de acordo com o código da **Listagem 6**. Na implementação do item que representa o botão *adicionando,* primeiramente estamos verificando se não existe nenhuma estrutura de campos já criada no *ClientDataSet* , o resultado sendo verdadeiro, iniciamos o processo zerando todo o conteúdo da propriedade *FieldDefs*. Após zerar o conteúdo iniciamos adicionando um primeiro campo chamado *Estado* que é do tipo *string*, com tamanho 2 e no último parâmetro informamos que o mesmo não se trata de um campo requerido. Seguindo adicionamos um campo chamado *NomeEstado* também do tipo *string*, sendo do tamanho 30 que também não é requerido. Finalizado a definição dos campos chamamos o método *CreateDataSet* para criar a estrutura física no componente e deixar disponível para utilização.

Fora do nosso *If* estamos alimentando os campos da estrutura já criada no *ClientDataSet*, adicionamos dados para os dois campos e chamamos o método *Post* para gravar os dados em memória. No *OnClick* do botão *Gravar* XML estamos gravando o conteúdo que está na memória do *ClientDataSet* na pasta onde está a aplicação executável no formato XML. E no *OnClick* do botão *Carregar XML* estamos carregando na memória do *ClientDataSet* o conteúdo do Arquivo XML que está em disco.

Para testar vamos executar a aplicação e clicar no botão *Adicionando* para adicionar dados. Depois clique no gravar para gerar um arquivo XML, feche a aplicação e execute-a novamente. Para ver se realmente gravamos dados no XML clique no botão *Carregar XML* e repare que os dados foram carregados do *DBGrid*, se quiser conferir o arquivo gerado em disco é só acessar a pasta onde foi gravado o projeto que o nosso arquivo estará lá dentro.

Tranformando um XML comum em DataPacket

Apesar do componente *ClientDataSet* ter a vantagem de fazer a geração do arquivo XML ele se baseia em um padrão proprietário por causa de sua estrutura, isso acontece visto que ele precisa sempre guardar o conteúdo do *Data "dados originais"* e do *Delta "dados modificados"*, com isso não conseguimos

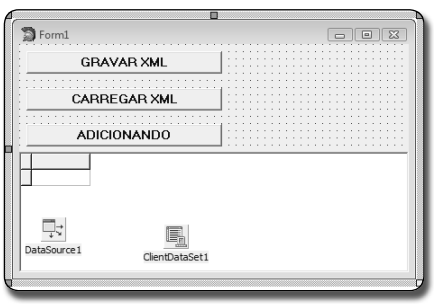


Figura 4. Design do projeto XMLDataPacket

Listagem 6. Implementação dos três Botões do projeto XMLDataPacket

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  { Botão Adicionando }
  if ClientDataSet1.FieldCount = 0 then
  begin
    ClientDataSet1.FieldDefs.Clear;
    ClientDataSet1.FieldDefs.Add('Estado',ftString, 2,False);
    ClientDataSet1.FieldDefs.Add('NomeEstado' ,ftString,30,False);
    ClientDataSet1.CreateDataSet;
  end;

  { Preencher o nosso ClientDataSet }
  ClientDataSet1.Append;
  ClientDataSet1.FieldByName('Estado').AsString:='SP';
  ClientDataSet1.FieldByName('NomeEstado' ).AsString:='São Paulo';
  ClientDataSet1.Post;
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  { Botão Gravar XML }
  ClientDataSet1.SaveToFile('States.xml',dfXML);
end;

procedure TForm1.Button3Click(Sender: TObject);
begin
  { Botão Carregar XML }
  ClientDataSet1.LoadFromFile('States.xml');
end;
```

por exemplo abrir diretamente nele um XML do padrão da **Listagem 1**, ou seja, dessa forma teríamos que manipular o XML manualmente para fazer leitura, escrita, pesquisa etc.

Pensando nisso temos no RAD Studio um componente chamado *XMLTransformProvider* cujo papel é fazer a transformação de um arquivo XML para o padrão *DataPacket*, ou seja, com um arquivo XML comum criamos arquivos de transformação chamados XTR e conseguimos mapear o arquivo na leitura para um *DataPacket* e abrir diretamente no *ClientDataSet*. E ainda melhor, conseguimos também fazer a volta. Com isso se alterarmos os dados ou incluirmos novos registros ao gravarmos, conseguimos gerar um arquivo XML não no formato *DataPacket* mais sim no formato XML de origem. Sinceramente, fantástico esse recurso. Mão na massa então pessoal.

Vamos criar uma nova aplicação acessando o menu *File>New>VCL Forms Application – Delphi for Win32* em seguida acesse o menu *File>Save All* e salve o projeto com o nome de *XMLTransform.dpr* e a *Unit* com o nome de *unitFrmTransform.pas*.

Em seguida adicione um *DbGrid*, um *DataSouce*, um *ClientDataSet*, um *XMLTransformProvider* e um *Button*. Selecione o *DbGrid* e aponte a propriedade DataSource para o *DataSource*, selecione o *DataSource* e aponte a propriedade *DataSet* para o *ClientDataSet* e para finalizar selecione o *ClientDataSet* e na propriedade *ProviderName* aponte para o *XMLTransformProvider*. Para esse exemplo vamos precisar ter um arquivo XML que não esteja no padrão *DataPacket*, então acesse o menu *File>New>Other*. E dentro do repositório acesse a aba *WebDocuments* e escolha a opção *XML File* e clique em *OK*. Adicione nesse arquivo gerado, o conteúdo que temos na **Listagem 1**. Em seguida acesse o menu *File>Save* e salve o arquivo com o nome de *dados.xml*.

Agora o próximo passo é gerar os arquivos de transformação através do *XML Mapper* do RAD Studio, acesse o menu *Tools>XML Mapper*, assim o mesmo será aberto sobre a IDE do RAD Studio. Em seguida acesse o menu *File>Open* e aponte para o arquivo *dados.xml* e clique em abrir. Em seguida pressione *Ctrl+A* para selecionar todos os campos do arquivo, o resultado você poderá conferir na **Figura 5**.

Após essa etapa vamos acessar o menu *Create>DataPacket from XML*, com isso a coluna *Selected Fields* terá os seus campos preenchidos para realizar o mapeamento. Agora acesse o menu *Create>Transformation* para ser gerado o primeiro arquivo de transformação. Após essa etapa acesse o menu *File>Save* e escolha a opção *Transformation*, aqui salvaremos um arquivo chamado *ToDp.xtr* que iremos guardar na pasta do nosso projeto. Gerado o arquivo de leitura agora precisamos gerar o arquivo para escrita. No item *Transform Direction* escolha a opção *DataPacket* to XML conforme **Figura 6**.

Para gerar o próximo arquivo de transformação acesse o menu *Create* novamente e escolha a opção *Transformation* para ser gerado o segundo arquivo de transformação, depois acesse o menu *File* e no item *Save* escolha a opção *Transformation* onde iremos salvar um arquivo chamado *ToXml.xtr* também na pasta do projeto.

Podemos agora fechar o *XML Mapper* e voltar para o RAD Studio para terminarmos de configurar a nossa aplicação. Selecione o componente *XmlTransformProvider* e na propriedade *XMLDataFile* selecione o arquivo *dados.xml*, depois acesse a propriedade *TransformationRead* e na sub propriedade *TransformationFile* aponte para o arquivo de leitura que é o *ToDp.xtr* e na propriedade *TransformationFile* do *Transformation Write* escolha o arquivo *ToXml.xtr*.

Feito isso já podemos acessar o nosso *ClientDataSet* e passar a propriedade *Active* dele para *True* e ver os dados em nosso *DbGrid*. Mas calma, ainda não acabou por aí, para finalizar vamos querer editar os dados no *DbGrid* e no botão gravar, salvar os dados em disco, e no formato XML de origem. Então, para finalizar dê um duplo clique no botão gravar, e escreva o código a seguir:

```
if ClientDataSet1.ApplyUpdates(0) = 0 then
  ShowMessage('Atualizado com
sucesso!!!');
```

Na verdade temos um *ApplyUpdates* com o parâmetro 0, que terá a responsabilidade de gravar os dados no XML e garantir que isso aconteça apenas se tudo der certo, ou seja, se ocorrer algum erro na gravação nada será gravado e dando tudo certo o resultado do mesmo será 0 e será exibido o nosso *ShowMessage* informando que os dados

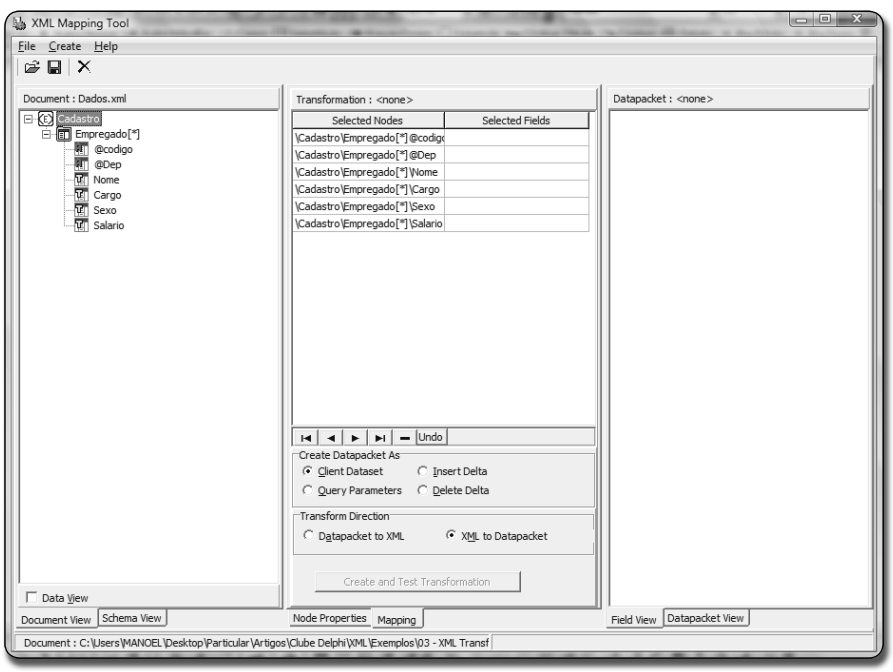


Figura 5. XML Mapper já com os primeiros campos selecionados

Figura 6. Configurando a direção da transformação

foram gravados. Depois é só executar a aplicação fazer um teste e o resultado você confere na **Figura 7**.

Manipulando XML com Interfaces

Outra forma de se manipular um arquivo XML utilizando técnicas relacionadas à Programação Orientada a Objetos é mapear um arquivo XML para interfaces e criar toda a iteração para ele: pesquisa, inserção, leitura e etc.

Através de métodos das interfaces, sem dúvida essa é uma maneira muito elegante, prática e ágil de manipular arquivos XML. Além de que podemos encapsular as interfaces em classes próprias para facilitar ainda mais o acesso. Aproveitando o tópico gostaria de comentar sobre um assunto muito discutido no site da DevMedia que foi uma solicitação da *ANS (Agência Nacional de Saúde)* para automatizar o processo entre as operadoras de plano de saúde e os prestadores, pois as exigências eram a implementação de um série de XMLs que seguiam diversos *schemas* de tipos simples e tipos complexos. Um dos meios mais simples de se implementar essa solução que foi chamado de *TISS*, sem dúvida, é o mapeamento dos arquivos para interfaces e encapsular os mesmos para classes. Para os que ainda se encontram perdidos ainda existe uma luz no fim do túnel, aproveitem esse conceito e revisem os seus códigos.

Continuando, para entender melhor como tudo funciona vamos criar uma nova aplicação no RAD Studio através do menu *File>New>VCL Forms Application Delphi for Win32*. Após criar a aplicação acesse o menu *File>Save All*. Para o projeto dê o nome de *XMLInterface* e para a *unit* dê o nome de *unitFrmInterface*. Em seguida adicione ao formulário um *TreeView*, um *XMLDocument*, 4 *Labels*, 4 *Edits* e 4 *Buttons*. Para os *Edits* dê os seguintes nomes *edtCodigo, edtNome, edtQuantidade* e *edtPreco* e para os *Buttons btnAtualizar, btnInserir, btnRemover* e *btnSalvar*. Também configure o *Caption* dos *Labels* e dos *Buttons* de acordo com a **Figura 8**.

Para esse exemplo vamos criar um novo arquivo XML, então acesse o menu *File>New>Other>Web Documents>XML File*. Em seguida acesse o menu *File>Save* e dê o nome para o arquivo de *produtos.xml* e em seguida vamos adicionar nele o conteúdo presente na **Listagem 7**.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Pauli que mostra como trabalhar com XMLMapper e XMLTransformProvider.

*www.devmedia.com.br/articles/viewcomp.asp?comp=5959&hl=*clientdataset**

Listagem 7. XML utilizado no projeto XMLInterface

```
<?xml version="1.0" encoding="UTF-8"?>
<Produtos>
<Produto codigo="1">
    <Nome>Introducao ao Delphi 2007</Nome>
    <Quantidade>6</Quantidade>
    <Preco>350</Preco>
</Produto>
<Produto codigo="2">
    <Nome>Delphi 2007 Aplicacoes for Win32</Nome>
    <Quantidade>7</Quantidade>
    <Preco>800</Preco>
</Produto>
<Produto codigo="3">
    <Nome>Delphi 2007 Aplicacoes Multicamadas</Nome>
    <Quantidade>8</Quantidade>
    <Preco>400</Preco>
</Produto>
</Produtos>
```

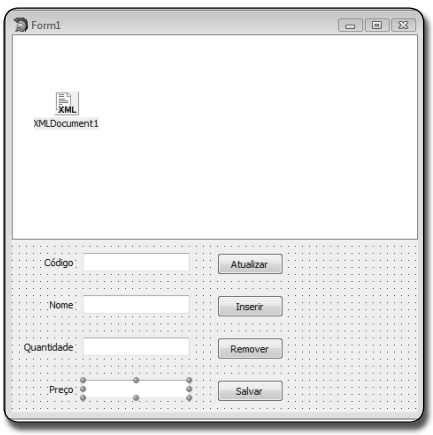


Figura 8. Interface do projeto XMLInterface

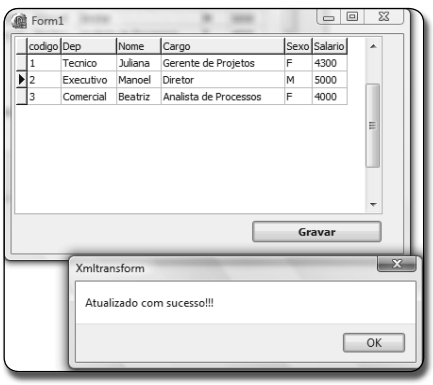


Figura 7. Aplicação com o XML Mapper em execução

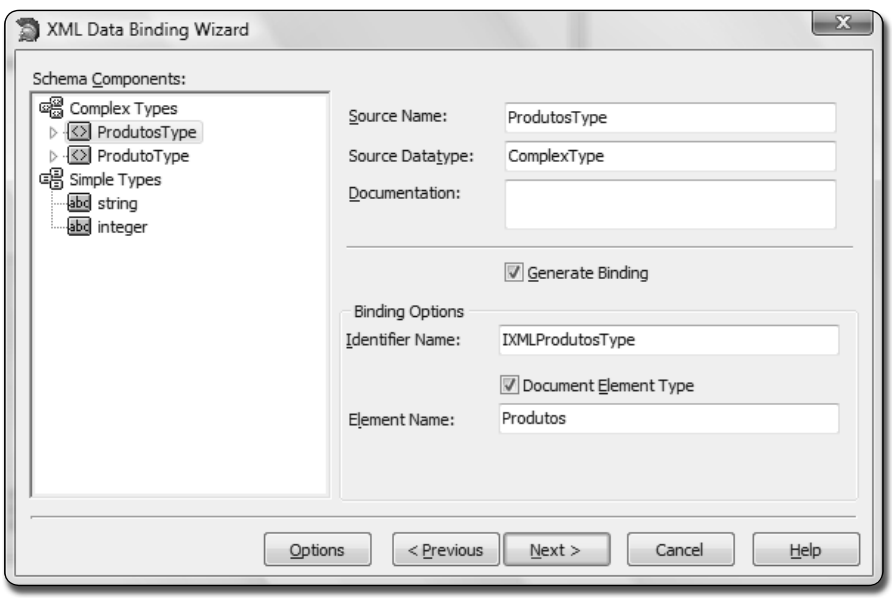


Figura 9. Mapeando o XML para Interface no XML Data Binding

O próximo passo é criar uma *unit* com as interfaces para manipular o arquivo XML em questão, para fazer isso vamos acessar o menu *File>New>Other>*XML*>XML Data Binding*. No item *Source* clique no botão reticências e escolha o nosso arquivo XML. Não se esqueça de configurar o filtro na *ComboBox* para XML e após selecionar o arquivo clique em *Abrir*. Na seqüência clique em *Next* e teremos uma tela com nomes sugeridos para os tipos e alguns detalhes adicionais conforme **Figura 9**.

Depois vamos clicar em *Next* e na seqüência em *Finish* onde será gerada uma *unit* com todas as definições de nossas estruturas já prontas. Em seguida acesse o menu *File>Save* e dê o nome de *unitProdutos*.

Seguindo o exemplo vamos acessar a *unit* do nosso formulário e na sessão *Private* adicione todas as declarações como a seguir:

```
Produto : IXMLProdutoType;
Estoque : IXMLProdutosType;
procedure AdicionaNo(NoXML :
   IXMLNode; Pai : TTreeNode);
procedure AtualizaTrVw;
```

Inicialmente temos a declaração de uma variável chamada *Produto* do tipo *IXMLProdutoType* que representa o item *Produto* do XML, seguindo temos uma chamada *Estoque* do tipo *IXMLProdutosType* que representa uma lista de *Produtos*. Temos o método *AdicionaNo* que já foi comentado em um outro exemplo nesse mesmo artigo e um método chamado *AtualizaTrVw* que terá o papel de dar um *Refresh* dos dados contidos em nosso *TreeView*.

Em seguida pressione *Ctrl + Shift + C*, lembrando que será necessário adicionar na cláusula *uses* da sessão *Interface* as units *unitProdutos* e *StrUtils*. Vamos agora programar nossos métodos, porém o método *AdicionaNo* é o mesmo já utilizado anteriormente aqui no artigo então a sua implementação é a mesma que se encontra na **Listagem 5**. Já o método *AtualizaTrVw* vamos implementar de acordo com o código da **Listagem 8**, nesse código podemos reparar que logo no início zeramos todo o conteúdo do *TreeView* e em seguida chamamos o método *AdicionaNo* passando como parâmetro a variável *Estoque* que representa a lista de todos os produtos. Como segundo parâmetro estamos passando *nil* que é o ponto do *TreeView* que vamos iniciar o nosso trabalho, como o conteúdo é *nil* estamos iniciando do ponto zero do *TreeView*.

Continuando a nossa implementação vamos selecionar o componente *XMLDocument* e na propriedade *FileName* vamos apontar para o arquivo XML criado que chamamos de *produtos.xml*, em seguida vamos dar um duplo clique em nosso formulário para implementar o código que estará no evento *OnCreate* dele de acordo com a **Listagem 9**, nesse código estamos atribuindo à variável estoque o retorno da função *GetProdutos* que devolverá a lista de todos os produtos contidos em nosso *XmlDocument* e em seguida vamos atualizar os dados no *TreeView*.

Carregada a lista de itens no *TreeView* o nosso próximo passo será implementar o evento *OnClick* do *TreeView*. A idéia é que, ao clicar em determinado item, a aplicação irá carregar os seus dados nos *Edits* permitindo o usuário, por exemplo, alterar os seus dados. Então vamos acessar o evento *OnClick* do *TreeView* e implementar ao código que se encontra na **Listagem 10**.

Declaramos inicialmente uma variável chamada *NoAtivo* do tipo *TreeNode* que irá representar o nó selecionado no *TreeView*. Em seguida estamos verificando através do método *AnsiContainsText* se no item selecionado do *TreeView* existe uma *string* chamada *código*. Se o retorno for verdadeiro estamos carregando na variável *NoAtivo* o nó selecionado do *TreeView* e em seguida estamos colocando na variável *Produto* que é correspondente ao item selecionado no *NoAtivo*. Finalizando, carregamos os dados que estão na variável para os componentes da tela. Se o resultado do *If* for falso estamos garantindo que os componentes da tela sempre estarão vazios e como último item estamos passando para a propriedade *Enabled* do botão *btnAtualizar* o estado da variável *Produto*.

Para concluir o nosso exemplo vamos precisar implementar o código dos nossos botões, que serão responsáveis

Listagem 8. Implementação do método AtualizaTrVw

```
procedure TForm1.AtualizaTrVw;
begin
  TreeView1.Items.Clear;
  AdicionaNo(Estoque, nil);
  TreeView1.Items.GetFirstNode.Expand(false);
end;
```

Listagem 9. Implementando o evento OnCreate do formulário

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  Estoque := GetProdutos(XMLDocument1);
  AtualizaTrVw;
end;
```

Listagem 10. Implementando o evento OnClick do TreeView

```
procedure TForm1.TreeView1Click(Sender: TObject);
var
  NoAtivo : TTreeNode;
begin
  if AnsiContainsText(TreeView1.Selected.Text,'codigo') then
  begin
    NoAtivo := TreeView1.Selected;
    Produto := Estoque.Produto[NoAtivo.Parent.IndexOf(NoAtivo)];
    with Produto do
    begin
      edtCodigo.Text := IntToStr(Codigo);
      edtNome.Text := Nome;
      edtQuantidade.Text := IntToStr(Quantidade);
      edtPreco.Text := CurrToStr(Preco);
    end;
  end
  else
  begin
    Produto := nil;
    edtCodigo.Clear;
    edtNome.Clear;
    edtQuantidade.Clear;
    edtPreco.Clear;
  end;
  btnAtualizar.Enabled := Assigned(Produto);
end;
```

por atualizar, gravar, remover e inserir dados no XML. As suas implementações podemos conferir na **Listagem 11**.

Iniciando pelo botão Atualizar, nesse botão pegamos o conteúdo dos componentes de tela e armazenamos na variável Produto e na seqüência chamamos o método AtualizaTrVw para dar um refresh em tela. Seguindo temos o OnClick do botão Insert que estamos chamando o método Add do item Estoque para adicionar um item produto a ele e recebendo o resultado na variável Produto. Em seqüência adicionamos o conteúdo aos campos e chamamos o método AtualizaTrVw para dar um refresh em tela novamente.

No *OnClick* do botão remover, estamos primeiramente definindo duas variáveis: uma chamada *NoAtivo* do tipo *TreeNode* para representar o *Nó* e outra chamada *I* do tipo *integer* para guardar o índice do item selecionado. Na primeira linha do código, estamos carregando o item selecionado na variável *NoAtivo*, em seguida estamos pegando o índice que a mesma representa para fazer e exclusão, então chamamos o método *Delete* e passamos como parâmetro o índice recuperado na variável *I*, e para atualizar os dados em tela chamamos o método *AtualizaTrVw*.

Finalizando os botões, implementamos o *OnClick* do botão *Salvar*, onde apenas chamamos o método *SaveToFile* do *XmlDocument* para atualizar os dados no arquivo em disco.

Depois de implementado o código já podemos executar a aplicação e desfrutar de todas as funcionalidades e facilidades obtidas nesse exemplo, além disso podemos conferir o resultado final em execução na **Figura 10**.

Listagem 11. Implementando o código dos Botões

```
procedure TForm1.btnAtualizarClick(Sender: TObject);
begin
  with Produto do
  begin
    Codigo := StrToInt(edtCodigo.Text);
    Nome := edtNome.Text;
    Quantidade := StrToInt(edtQuantidade.Text);
    Preco := StrToInt(edtPreco.Text);
  end;
  AtualizaTrVw;
end;
procedure TForm1.btnInserirClick(Sender: TObject);
begin
  Produto       := Estoque.Add;
  with Produto do
  begin
    Codigo := 0;
    Nome := '<Nome>';
    Quantidade := 0;
    Preco := 0;
  end;
  AtualizaTrVw;
end;
procedure TForm1.btnRemoverClick(Sender: TObject);
var
  NoAtivo : TTreeNode;
  I : integer;
begin
  NoAtivo := TreeView1.Selected;
  I := NoAtivo.Parent.IndexOf(NoAtivo);
  Estoque.Delete(i);
  AtualizaTrVw;
end;
procedure TForm1.btnSalvarClick(Sender: TObject);
begin
  XMLDocument1.SaveToFile;
end;
```

Conclusão

Através deste artigo podemos concluir, que sem dúvida, os arquivos XML são mais poderosos do que pensávamos, e o RAD Studio está muito bem posicionado no que diz respeito à manipulação, geração, exibição etc. Percebemos que a facilidade que se tem para manipular esse tipo de arquivo é inacreditável, pois nos facilita muito o dia-a-dia, além de conseguirmos vislumbrar diversos trabalhos complexos que tínhamos, agora com muito mais facilidade. Enfim pessoal, vamos definitivamente deixar o "TXT" de lado e migrar tudo o que temos para XML, além do que, hoje em dia, portabilidade é tudo. Vou ficando por aqui e nos encontramos em um próximo artigo. Até o próximo. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

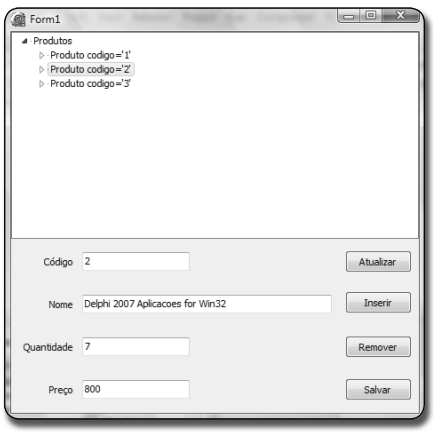
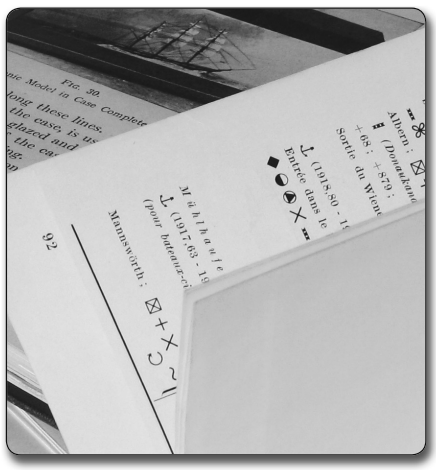


Figura 10. Projeto XMLInterface em execução



Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

Criação Automatizada de Formulários

Agilize o processo de criação de cadastros usando técnicas avançadas de automatização de tarefas

Resumo DevMan

Sabemos que no dia-a-dia de qualquer software house ou mesmo desenvolvedores autônomos, tempo é dinheiro e agilizar o processo de desenvolvimento é sempre bem-vindo. Com certeza as telas que mais demandam tempo em um projeto real são as de cadastro. Geralmente são telas comuns e quase com as mesmas funções. Mesmo assim gastamos grande parte do nosso desenvolvimento fabricando tais mecanismos. Por que não automatizar tudo isso? Criar janelas de cadastro genéricas onde o próprio sistema as cria?

Pois bem, baseados nisso faremos um exemplo bem completo onde faremos a criação automática de telas de cadastro automatizando e agilizando esse processo em nossos sistemas.

Nesse artigo veremos

• Como criar uma classe capaz de gerar formulários automaticamente;

• Inclusão automática de controles em tela;

• Técnicas avançadas de automatização de tarefas.

Qual a finalidade

• Com as técnicas apresentadas é possível agilizar o processo de criação de telas de cadastro.

Quais situações utilizam esses recursos?

• Podemos aplicar esses recursos em praticamente todo tipo de sistema, porém o maior foco são softwares que necessitam de telas de cadastro.

Gustavo Chaurais

(gchaurais@gmail.com)

é Borland Delphi 7 Advanced Certified, Borland Delphi 2005 for Win32 Certified, Borland Delphi 2006 for Win32 Certified e Borland Delphi Instructor Certified. Foi palestrante das três últimas edições da Borland Conference Brasil e de outros grandes eventos nacionais. Hoje, trabalha com cursos, consultorias e atua como Software Engineer do projeto InstallAnywhere para a empresa norte-americana Macrovision/Acresso.

A partir do momento em que iniciamos a construção de um projeto de grande porte, temos em mente a dificuldade que será mantermos a compatibilidade entre cada formulário da interface com o usuário final. Mais do que isso, sabemos do tempo a ser investido na confecção de cada um deles separadamente.

Entretanto, desenvolvedores um pouco experientes certamente já pensaram: "se estes formulários de cadastro são tão básicos, porque não existe algo que automatize sua criação?". Pois, essa tarefa só nos toma tempo e stress na hora de verificar se tudo está compatível.

Mesmo no caso de estarmos utilizando uma estrutura hierárquica bem estruturada de formulários, podemos encontrar problemas desse tipo. Por exemplo, se

quisermos colocar todos os campos *TEdit* para edição em letras maiúsculas, teremos de ir tela por tela alterando todos os componentes. Ou optarmos por adicionar um algoritmo na classe base que varra os componentes e faça as alterações em *runtime*. Isso é bastante trabalhoso. A situação piora quando queremos acabar com abreviações em *labels*, ou verificar espaços entre componentes, ou mudar a classe base de um formulário etc.

Através de um aplicativo de exemplo bastante prático, a solução apresentada é a da construção de formulários em *runtime*, ou seja, com apenas algumas linhas de código, criamos telas para cadastro de informações básicas de nosso projeto. Assim, não precisaremos investir nosso tempo em tarefas cotidianas, facilmente automatizáveis. Além disso, estas classes base podem ser facilmente acopladas aos *frameworks* para desenvolvimento de aplicações que utilizamos em nossas empresas.

A classe base do formulário

Antes de iniciar a automação propriamente dita, vamos criar uma classe base, da qual todos os formulários de cadastro de nossa aplicação herdarão. Assim, todas as operações de cadastro terão um ponto em comum.

> **Nota:** Se você já possui um projeto em andamento, tente, em vez de criar outra classe, utilizar a classe base dos formulários de seu próprio sistema. Com algumas pequenas alterações, certamente o resultado será o mesmo e se adaptará ao seu contexto.

Crie um novo projeto chamado *AutoFormTestApp.dpr* e renomeie o formulário gerado para *fmBaseForm*, salvando-o em seguida como *BaseForm.pas*. Baseando-se na **Figura 1**, adicione e configure os seguintes componentes. Inclua um *SQLConnection*(*conAutoForm*) e configure sua propriedade *LoginPrompt* para *False*. Também da paleta *dbExpress* inclua um *SQLQuery*(*sqAutoForm*) e conecte-o ao *conAutoForm* pela propriedade *SQLConnection*. Acesse a paleta *Data Access* e arraste um *DataSetProvider*(*dspAutoForm*) configurando sua propriedade *DataSet* para *sqAutoForm*. Em seguida arraste um *ClientDataSet*(*cdsAutoForm*) conectando-o ao *DataSetProvider* pela propriedade *ProviderName*. Por fim inclua um *DataSource*(*dsAutoForm*) com a propriedade *DataSet* igual a *cdsAutoForm*. Com isso criamos uma conexão com nosso banco de dados.

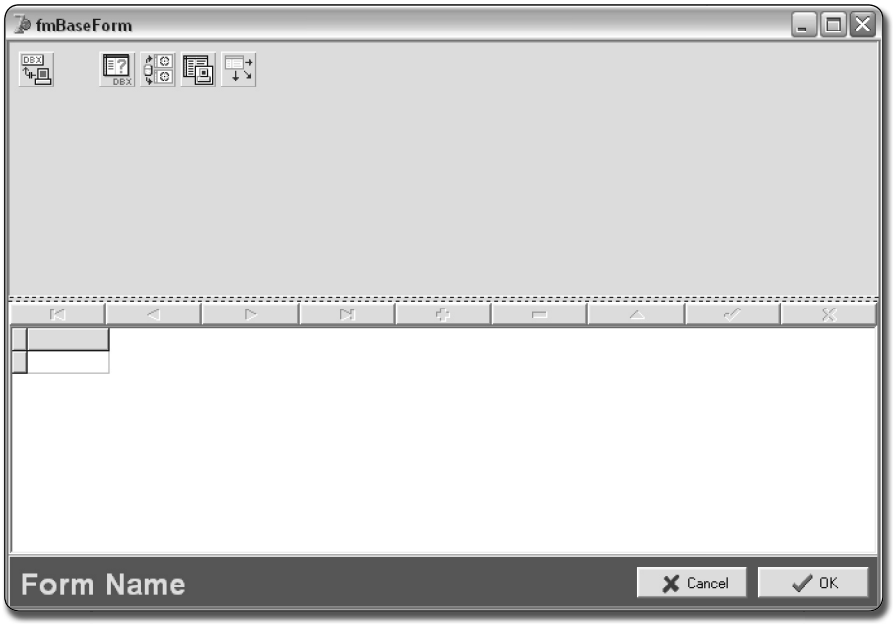


Figura 1. Configuração dos componentes do formulário base

Agora faremos a inclusão de componentes visuais começando por um *Panel*(*pnlControls*). Modifique sua propriedade *Align* para *alTop*. Insira um novo *Panel*(*pnlWindowNav*) com a propriedade *Align* igual a *alBottom*. Em seguida arraste um *Splitter*(*splGrid*) da paleta *Additional* com sua propriedade *Align* igual a *alTop*. Esse componente será útil para que possamos redimensionar a área de visualização dos controles. Insira um novo *Panel*(*pnlGrid*) como *alClient*. Dentro do *pnlWindowNav* inclua um *Label*(*lblFormName*) e *Caption* igual a *Form Name*, um *BitBtn*(*btnOk*) com *Caption* igual a *Ok*, *Kind* igual a *bkOk* e *Default True*. Ainda no *pnlWindowNav* coloque outro *BitBtn*(*btnOk*) com *Caption* igual a *Cancel* e *Kind* igual a *bkCancel*.

Já dentro do *pnlGrid* insira um *DBNavigator*(*dbnAutoForm*) com o seu *DataSource* apontado para *dsAutoForm* e *VisibleButtons* desmarque (*False*) o item *nbRefresh*. Modifique sua propriedade *Align* para *alTop*. Para finalizar inclua dentro do *pnlGrid* um *DBGrid*(*dbgAutoForm*). Modifique seu *Align* para *alClient*, *DataSource* para *dsAutoForm* e *Options* desmarque (*False*) a opção *dbTabs*.

Com isso conseguimos montar um layout semelhante a **Figura 1.** Fiz ainda algumas modificações nas propriedades de fonte do *Label lblFormName* bem como o fundo do *Panel* onde ele está alocado, ou seja, fica a seu critério modificar a parte visual como desejar em nossa aplicação.

Para a configuração do banco de dados, utilize o *SGBD* de sua preferência e maior familiaridade. Apenas lembre-se de configurar o componente *conAutoForm* (*TSQLConnection*) corretamente. Preferivelmente, crie um novo banco de dados para o exemplo. Serão necessárias 6(seis) tabelas em nosso banco de dados. Na **Figura 2** fiz questão de descrever as propriedades de cada tabela que deverá ser criada. Não esqueça que as tabelas *Produto*, *Venda* e *Venda_Produto* deverão conter *Foreign Key*, ou seja, chaves estrangeiras nos campos:

- *Produto*: campos *ID_MARCA*;
- *Venda: ID_VENDEDOR e ID_CLIENTE;*
- *Venda_Produto: ID_PRODUTO.* Esse também deverá fazer parte da *Chave Primária* da tabela.

Não entrarei em detalhes sobre a criação do banco, tabelas, chaves primárias e/ou chaves estrangeiras por não fazer parte do escopo desse artigo.

Como você pode ver, esta será uma aplicação exemplo clássica de *venda de produtos*. Vamos então incrementar um pouco nosso formulário base.

Codificando o formulário

Vamos iniciar a codificação do nosso formulário padrão começando pelo evento *OnShow*. Manipule o evento *citado do Form* incluindo as linhas de programação a seguir:

```
lblFormName.Caption :=
  'Cadastro de ' + GetDisplayName;
  Self.Caption := lblFormName.Caption;
cdsAutoForm.Open;
```

Perceba que a codificação é bem simples. Apenas modificamos o *Caption* do *form* concatenando a frase *Cadastro de* mais o método *GetDisplayName* que veremos logo a seguir. Em seguida nós abrimos o *ClientDataSet cdsAutoForm*. Declare agora, na seção *Protected*, da classe *TfmBaseForm* as seguintes funções:

```
function GetKeyField: string; virtual;
  abstract;

function GetDisplayName: string; virtual;
  abstract;
```

Ou seja, estamos criando dois métodos abstratos, os quais serão obrigatoriamente sobrescritos nas sub-classes. *GetKeyField* deverá retornar o nome do campo com a chave principal da tabela em questão. Já *GetDisplayName* retornará o nome da informação que estará sendo cadastrada, por exemplo, *Produtos em Estoque*. Quando o formulário aparecer, a informação *Cadastro de X* aparecerá no *label* e na barra de títulos, e o *Dataset* será aberto caso esteja fechado em *design time*.

> **Nota:** Para facilitar, estamos partindo do princípio de que todas as tabelas terão uma única chave primária, a menos que sejam tabelas auxiliares. Você poderia facilmente adaptar o código para trabalhar com chaves compostas.

Listagem 1. Declaração das funções e da seção Protected

```
type
  TForm1 = class(TForm)
  private
    { Private declarations }
  public
    { Public declarations }
  protected
    function GetKeyField: string; virtual; abstract;
    function GetDisplayName: string; virtual; abstract;
  end;
```

Listagem 2. Evento BeforePost do ClientDataSet

```
procedure TfmBaseForm.cdsAutoFormBeforePost(DataSet: TDataSet);
var
  i: Integer;
begin
  if cdsAutoForm.FieldByName(GetKeyField).IsNull then
    cdsAutoForm.FieldByName(GetKeyField).AsInteger := GetNextKeyValue;

  for i := 0 to cdsAutoForm.FieldCount - 1 do
  begin
    if (cdsAutoForm.Fields[i].FieldKind = fkData) and
      cdsAutoForm.Fields[i].Required and (cdsAutoForm.Fields[i].IsNull or
        (cdsAutoForm.Fields[i].AsString = '')) then
      begin
        MessageDlg('Campo requerido não foi preenchido: ' +
          cdsAutoForm.Fields[i].DisplayLabel, mtError, [mbOk], 0);
        Abort;
      end;
  end;
end;
```

Cliente

#	PK	Field Name	Field Type	Size	Not Null
	1	ID_CLIENTE	INTEGER		☒
		NOME	VARCHAR	50	☒
		IDADE	INTEGER		☒
		SEXO	CHAR	1	☒

Vendedor

#	PK	Field Name	Field Type	Size	Not Null
	1	ID_MARCA	INTEGER		☒
		NOME	VARCHAR	20	☒
		CONTATO	VARCHAR	50	☒

Marca

#	PK	Field Name	Field Type	Size	Not Null
	1	ID_VENDEDOR	INTEGER		☒
		NOME	VARCHAR	50	☒
		MATRICULA	INTEGER		☒

Produto

#	PK	Field Name	Field Type	Size	Not Null
	1	ID_PRODUTO	INTEGER		☒
		NOME	VARCHAR	20	☒
		ID_MARCA	INTEGER		☒
		QUANTIDADE_ESTOQUE	INTEGER		☒

Venda

#	PK	Field Name	Field Type	Size	Not Null
	1	ID_VENDA	INTEGER		☒
		DATA	DATE		☒
		ID_VENDEDOR	INTEGER		☒
		ID_CLIENTE	INTEGER		☒

Venda_Produto

#	PK	Field Name	Field Type	Size	Not Null
	1	ID_VENDA	INTEGER		☒
		ID_PRODUTO	INTEGER		☒
		QUANTIDADE	INTEGER		☒

Figura 2. Estrutura das tabelas do banco de dados

Perceba que não temos uma seção *Protected* no formulário, porém podemos facilmente criá-la apenas declarando-a na seção *Type* do *form* como podemos ver na **Listagem 1**.

Agora será necessário manipular o evento *OnClick* do botão *btnOk* apenas incluindo uma chamada ao método *ApplyUpdates* do *ClientDataSet* como segue:

```
cdsAutoForm.ApplyUpdates(0);
```

Isso fará com que as alterações realizadas na tela sejam aplicadas ao SGBD. E, finalmente, manipule o evento *BeforePost* do *TClientDataSet* conforme a **Listagem 2**.

Neste trecho de código, estamos primeiramente verificando se o valor correspondente à chave primária é nulo. Caso seja, atribuímos o valor correto a ele (será mostrado adiante). Após isso, varremos todos os *TFields* do *TClientDataSet* procurando campos do tipo *Data* (não calculados ou *lookup*) que sejam requeridos (*NOT NULL* trazido do banco de dados) e que estejam com o valor nulo ou vazio. No caso de haverem tais campos, mostramos uma mensagem de erro para o usuário e abortamos a operação de *Post*.

Talvez você esteja se perguntando: "por que utilizar a função *MessageDlg* em vez de apenas disparar uma exceção com

raise?". Essa é uma longa discussão, mas, particularmente, penso que exceções seriam estados anormais, nos quais o programa pode se encontrar. Porém, estes seriam casos em que o próprio executável se coloca por alguma situação não tratada no momento da programação. Por exemplo, uma tentativa de acesso a uma região não alocada na memória resulta em um *Access Violation*. Exceções do tipo divisão por zero ou erro ao tentar passar de *string* para *integer* deveriam ser tratadas com um bloco *try..except*. Para resumir, exceções devem ser disparadas para *desenvolvedores*. Mensagens de erro é que são para *usuários finais*. E, no caso anterior, campos não preenchidos na hora do preenchimento de um formulário não são, de maneira alguma, erros do programador. Nada nos impede, também, de implementar um tratamento de exceções genérico que capture-as e mostre-as de maneira mais amigável ao usuário final.

Aqui você também poderia escrever um tratamento de validação de campos bem mais elaborado. Pois, o apresentado é bastante simples. Fica este como matéria para próximos artigos.

Vamos agora pensar um pouco. Como poderia ficar nossa função que retorna o próximo valor da chave primária da tabela? Uma primeira resposta seria a utilização de algum artifício proveniente do próprio SGBD. No entanto, teríamos de escrever diversas classes para mapear todos os *sequences*, *generators* e outros utilitários para nosso sistema. Não seria difícil, porém bastante trabalhoso e geraria um grande volume de código, o que dificultaria sua manutenção. Uma outra solução bem mais simplória, seria chamar uma instrução do tipo *select max(ID_TABELA) + 1 from TABELA*. Alguns poucos casos teriam de ser tratados, pois, alguns bancos de dados retornam *NULL*, caso a função *MAX* retorne *NULL*. Outros, retornam *0*. Isso é, contudo, bastante simples de se tratar, perto do problema que deveremos enfrentar aqui.

Vamos recordar o funcionamento do *TClientDataSet*. Sua principal função é fazer *cache* dos dados (*data*) e suas modificações (*delta*) no lado cliente da aplicação.

Listagem 3. Código para geração da Chave Primária

```
function TfmBaseForm.GetNextKeyValue: Integer;
var
  AClone: TClientDataSet;
begin
  Result := 1;

  if not cdsAutoForm.IsEmpty then
  begin
    AClone := TClientDataSet.Create(Self);
    try
      AClone.CloneCursor(cdsAutoForm, False);
      AClone.IndexFieldNames := GetKeyField;
      AClone.Last;
      Result := AClone.FieldByName(GetKeyField).AsInteger + 1;
    finally
      AClone.Free;
    end;
  end;
end;
```

Por isso, caso o usuário pretenda inserir dois ou mais registros, como ainda não teremos chamado o método *ApplyUpdates* (em nosso caso, isso só acontece ao se clicar no botão *OK*), cada vez que um registro é salvo, chamaremos a função SQL anterior e esta retornará sempre o mesmo valor. Por exemplo, se tivermos três registros na tabela com seus valores de chaves primária iguais a 1, 2 e 3, ao adicionarmos um registro ao *TClientDataSet*, seu *ID* será configurado com 4. Contudo, como isso ainda não foi aplicado, na próxima adição, em vez de 5, ainda receberemos 4.

Antes de continuar a ler, tente encontrar sozinho uma solução para este problema. Existem algumas. Umas mais e outras menos complexas. A que iremos adotar pode ser vista na **Listagem 3**.

Utilizamos como base os dados já modificados do *TClientDataSet* que estamos utilizando (*cdsAutoForm*). Mas, como neste momento o estado do *DataSet* será provavelmente *dsInsert*, em vez de *dsBrowse*, não poderemos modificar seu cursor para navegar nos registros e procurar o maior valor para a chave. Por este motivo, utilizaremos um *clone* com os mesmos dados já modificados. Neste *clone*, o estado do *DataSet* ainda estará como *dsBrowse* e, portanto, podemos navegar tranquilamente. Devemos então, ou procurar o maior valor com um *loop* ou, simplesmente, configurar o índice que irá reger os dados para o campo com a chave primária (pois este pode ter sido modificado) e pegar seu último valor. Como estes valores serão únicos (*UNIQUE*) e não nulos (*NOT NULL*), teremos a certeza de estar pegando o último valor. A este, adicionamos um e, conseqüentemente, temos o próximo valor para a chave primária.

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Pauli que mostra o que exatamente são Data e Delta assim como trabalhar com eles.

*www.devmedia.com.br/articles/viewcomp.asp?comp=5717&hl=*clientdataset**

Nota do DevMan

Há momentos em que necessitamos fazer pesquisa de determinados dados dentro de um *ClientDataSet* já carregado para memória. Podemos simplesmente fazer essa pesquisa utilizando alguns métodos bastante conhecidos, como o Locate do TDataSet por exemplo. Porém, há um problema quando fazemos isso: perdemos a referência ao registro que já estava posicionado.

Para contornar esse problema, podemos simplesmente criar um novo ClientDataSet e "clonar" o cursor, assim teremos uma cópia do DataSet e poderemos efetuar quantas pesquisas quisermos. Para isso basta fazermos uma chamada ao método CloneCursor do próprio ClientDataSet. Veja sua declaração original:

```
procedure CloneCursor(Source:
TCustomClientDataSet; Reset: Boolean;
KeepSettings: Boolean);
```

***Source*: é o ClientDataSet a ser clonado;**

Reset e KeepSettings são usados para determinar como serão usados os atributos *Filter, Filtered, filterOptions, onFilterRecord, IndexName MasterSource, MasterFields, ReadOnly, RemoteServer e ProviderName.*

Apenas mantenha *False* em nosso exemplo.

E, falando sobre índices, manipule o evento *OnTitleClick* do *dbgAutoForm* incluindo a seguinte linha:

```
if Column.Field.FieldKind = fkData then
  cdsAutoForm.IndexFieldNames :=
    Column.FieldName;
```

Desta maneira, quando o usuário clicar no título de alguma coluna (que não seja calculada ou *lookup*), o *cdsAutoForm* (e, por conseqüência, o *grid*) se ajustarão ao novo índice.

Formulários criados automaticamente

Agora que já temos a classe base para todos os formulários de cadastro de nossa aplicação, vamos finalmente iniciar a construção da classe que criará *forms* dinamicamente. Adicione então uma nova classe que herde *TfmBaseForm* usando o menu *File>New>Other>AutoFormTestApp>fmBaseForm*. Nomeie este formulário como *fmAutoForm* e salve a nova *unit* como *AutoForm.pas*. Sua utilização será bastante simples. Criaremos uma instância dessa classe e chamaremos métodos para dizer quais campos gostaríamos que estivessem lá. O resto será feito internamente. Declare dois construtores na seção *public* de sua classe como podemos ver a seguir:

```
constructor Create(AOwner: TComponent;
  const TableName, TableDisplayName,
KeyField, Filter: string);
  reintroduce; overload;
constructor Create(AOwner: TComponent;
  const SQL, TableDisplayName,
KeyField: string); reintroduce; overload;
```

Note a diretiva *reintroduce*. Ela fará com que a mensagem de *warning*, dizendo que estamos escondendo o construtor virtual, não apareça. E, de fato, queremos escondê-lo para que somente os novos sejam utilizados. Os valores a serem passados são:

- O *owner* do componente, a ser utilizado pelo *construtor* da superclasse;
- O nome da tabela ou uma instrução *SQL select*;
- Um nome mais legível para esta tabela;
- O nome do campo com a chave primária;
- Um filtro (por exemplo, *CAMPO_A > 10 and CAMPO_B is null*), caso não tenha sido passada uma instrução *SQL*.

Chamaremos o construtor que recebe a função *SQL* de *principal* e o outro de *secundário* (pois este utilizará o principal) daqui por diante. Implemente o construtor principal como na **Listagem 4**.

Sua primeira instrução será chamar o construtor da classe base para que o formulário seja criado corretamente. As segundas e terceiras instruções armazenarão os valores da chave primária e do nome legível da tabela a ser utilizado na classe base. Por isso, declare os seguintes atributos na seção *private*:

```
FTableDisplayName: string;
FKeyField: string;
```

Declare também (na seção *protected*) e implemente os métodos *abstract* da classe base (não esqueça da diretiva *override*). Veja as implementações na **Listagem 5**.

A quarta instrução configura a propriedade *SQL* do *TSQLQuery* como sendo a informada ao construtor. Já a quinta e a sexta, são necessárias, pois, como nem todos os *TFields* serão adicionados ao *TClientDataSet*, precisamos de um outro *TDataSet* que armazene somente os metadados de todos os campos da tabela, para futura consulta. Portanto, adicione um *TSimpleDataSet* ao seu formulário, nomeando-o *sdsMetadata* e configure suas propriedades *Connection* (apontando para *conAutoForm*) e *PacketRecords* (para 0). Esta última fará com que nenhum dado seja populado no *TSimpleDataSet*, apenas metadados.

As duas últimas instruções são responsáveis pela criação do *TField* correspondente à chave primária da tabela. Nossa intenção será a de fazer com que o componente *TDataSetProvider* gere uma instrução *SQL* bastante "limpa", colocando apenas a chave primária na clausula *WHERE*. E este é o motivo de estarmos adicionando e configurando os campos também no *TSQLQuery*. Caso contrário, não ficaria tão claro quais *TFields* deveriam ser adicionados à clausula *WHERE*

Listagem 4. Código Create da classe

```
constructor TfmAutoForm.Create(AOwner: TComponent; const SQL,
  TableDisplayName, KeyField: string);
begin
  inherited Create(AOwner);
  FKeyField := KeyField;
  FTableDisplayName := TableDisplayName;
  sqAutoForm.SQL.Text := SQL;
  sdsMetaData.DataSet.CommandText := SQL;
  sdsMetaData.Open;

  AddKeyField(KeyField, sqAutoForm);
  AddKeyField(KeyField, cdsAutoForm);
end;
```

Listagem 5. Código de GetKeyField e GetDisplayName

```
function TfmAutoForm.GetKeyField: string;
begin
  Result := FKeyField;
end;

function TfmAutoForm.GetDisplayName: string;
begin
  Result := FTableDisplayName;
end;
```

Listagem 6. Código para adição de campos

```
procedure TfmAutoForm.AddKeyField(const KeyField: string; const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
begin
  AddedKeyField := TIntegerField.Create(ToDataSet);
  AddedKeyField.FieldName := KeyField;
  AddedKeyField.Visible := False;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;
end;
```

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Pauli que mostra como utilizar PacketRecords em sua aplicação.

*www.devmedia.com.br/articles/viewcomp.asp?comp=5895&hl=*clientdataset**

pelo *provider*. Portanto, os componentes *TSQLQuery* e *TClientDataSet* terão os mesmos campos e, estes, corresponderão exatamente aos campos adicionados pelo usuário da classe.

Para concretizar nossa intenção de manter o *SQL* gerado sempre "limpo", modifique a propriedade *UpdateMode* do *dspAutoForm* (na classe *TfmBaseForm*) para *upWhereKeyOnly*, ou seja, na seção *WHERE* só será adicionada a chave. Agora, implemente a função *AddKeyField* como segue na **Listagem 6**.

Neste caso, estamos criando um componente *TField* do tipo inteiro que representará a chave primária da tabela. Este método será chamado pelo construtor para os dois *TDataSets* mencionados anteriormente. Além disso, precisamos implementar um outro método, o *SetUpWithReadMetaData* como na **Listagem 7**. Nesse código configuramos o campo adicionado, modificando seu *Tamanho*(*Size*), se é *Obrigatório* ou não (*Required*) e seu *ProviderFlags*.

Neste ponto, finalmente, se você tentar compilar seu projeto, obterá sucesso. Este método será vastamente utilizado por nossa classe para se obter propriedades de um *TField* que está sendo adicionado. Como todos os *TFields* disponíveis estão em *sdsMetadata*, simplesmente configuramos as propriedades do campo que está sendo criado de acordo. Além disso, se este for a chave primária, adicionamos *pfInUpdate* e *pfInKey* para nos assegurar de que será acrescentado à seção *WHERE* do *SQL* de atualização.

O outro construtor tem implementação bem simples, veja a **Listagem 8**.

Montamos a instrução *SQL* nós mesmos (para facilitar o trabalho de quem utilizará a classe) e chamamos o construtor principal.

Até agora, nosso exemplo ainda não tem muita utilidade. Se você reparar, ainda não rodamos nem uma vez nossa aplicação. Então, vamos finalmente a algo um pouco menos abstrato.

Declare e implemente os seguintes métodos *AddStringField* e *InternalAddStringField* conforme a **Listagem 9**.

O primeiro, que ao contrário do segundo deverá estar na seção *public*, será chamado pelo usuário da classe para que um campo do tipo *string* seja adicionado. Esse será responsável por chamar o segundo duas vezes, uma para o *sqAutoForm* e outra para o *cdsAutoForm*. Conforme mencionado anteriormente, isto é apenas para que a instrução *SQL* seja gerada como esperamos.

O segundo criará um *TField* do tipo *string* e o configurará corretamente. Note que são passados *DisplayFieldName* e *DisplayWidth*, para que o componente tenha uma leitura mais fácil ao ser adicionado na tela. E após a adição dos *TFields* nos dois *TDataSets*, o método *CreateStringControl* é chamado para que um componente compatível seja criado na tela e o usuário possa manipulá-lo. Sua implementação pode ser vista na **Listagem 10**.

Note que um controle *TDBEdit* é criado e configurado, mas este ainda não é adicionado a tela. Isto é papel do método *AddControl*, talvez o maior e mais trabalhoso do exemplo inteiro. Mas antes de vermos sua implementação, declare a seguinte constante:

```
const
  MARGIN = 10;
```

Listagem 7. Código SetUpWithReadMetaData

```
procedure TfmAutoForm.SetUpWithReadMetaData(const Field: TField);
begin
  Field.Size := sdsMetadata.FieldByName(Field.FieldName).Size;
  Field.Required := sdsMetadata.FieldByName(Field.FieldName).Required;
  if Field.FieldName = FKeyField then
    Field.ProviderFlags := [pfInWhere, pfInUpdate, pfInKey]
  else
    Field.ProviderFlags := [pfInUpdate];
end;
```

Listagem 8. Construtor da classe

```
constructor TfmAutoForm.Create(AOwner: TComponent; const TableName,
  TableDisplayName, KeyField, Filter: string);
var
  ASql: string;
begin
  ASql := 'select * from ' + TableName;

  if Filter <> '' then
    ASql := ASql + ' where ' + Filter;

  Self.Create(AOwner, ASql, TableDisplayName, KeyField);
end;
```

Listagem 9. Métodos AddStringField e InternalAddStringField

```
procedure TfmAutoForm.AddStringField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer);
begin
  InternalAddStringField(FieldName, DisplayFieldName, DisplayWidth, sqAutoForm);
  InternalAddStringField(FieldName, DisplayFieldName, DisplayWidth, cdsAutoForm);
end;
procedure TfmAutoForm.InternalAddStringField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer; const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
begin
  AddedKeyField := TStringField.Create(ToDataSet);
  AddedKeyField.FieldName := FieldName;
  AddedKeyField.DisplayLabel := DisplayFieldName;
  AddedKeyField.DisplayWidth := DisplayWidth;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;

  if ToDataSet = cdsAutoForm then
    CreateStringControl(AddedKeyField);
end;
```

Listagem 10. Implementação do código CreateStringControl

```
function TfmAutoForm.CreateStringControl(const Field: TField): TControl;
var
  AControl: TDBEdit;
begin
  AControl := TDBEdit.Create(Self);
  AControl.Width := Field.DisplayWidth * 6;
  AControl.DataSource := dsAutoForm;
  AControl.DataField := Field.FieldName;
  AddControl(AControl, Field.DisplayLabel);
  Result := AControl;
end;
```

Esta será a margem (em *pixels*) para a disponibilização dos componentes na tela. Declare também (na seção *private*) as variáveis auxiliares:

```
FCurHeight, FCurWidth, FMaxHeight: Integer;
```

Inicializando-as no construtor principal:

```
FCurHeight := MARGIN;
FCurWidth := MARGIN;
FMaxHeight := MARGIN;
```

Finalmente, vamos ao *AddControl*. Crie uma nova *procedure* com o nome mencionado e crie seu código conforme a **Listagem 11**.

Nesse ponto, seu código deve estar compilando novamente. A implementação deste método não é tão complicada quanto parece. Vamos primeiro entender as três variáveis auxiliares que declaramos anteriormente. Os componentes serão adicionados da direita para a esquerda e, quando completar a linha, inicia-se o preenchimento da próxima. *FCurHeight*, então, será um cursor que armazenará a posição vertical atual no formulário. *FCurWidth* terá a mesma função, mas para a posição horizontal. E *FMaxHeight* armazenará o limite vertical máximo que um controle chegou a alcançar na tela. A diferença entre *FCurHeight* e *FMaxHeight* então será: o primeiro será responsável por conter o valor do início (*top*) da linha atual. O segundo terá o valor mais baixo (*bottom*) que o maior componente (verticalmente) naquela linha conseguiu atingir.

Começamos a explicação do método diretamente na criação do *TLabel*, pois as constantes são auto-explicativas. O resultado final será um controle com um *label* rotulando-o logo acima. E como o controle já foi criado (este é passado por parâmetro), podemos comparar seu tamanho com o do *label* para ver se o controle não ficará menor do que o *TLabel* (o que não ficaria esteticamente bonito). Por isso, corrigimos a largura (*width*) do controle.

A comparação na seqüência tem a finalidade de verificar a necessidade da mudança de linha. Se a posição horizontal atual, somada à largura do componente e à margem for maior do que a largura do espaço disponível, a linha atual passa a ser a próxima.

Configuramos então os componentes, verificando se o *label* precisa ou não ser mostrado e reconfiguramos *FMaxHeight*, se necessário. Para finalizar, o valor de *FCurWidth* também é reajustado. Feito isso, você pode adicionar diversos outros métodos para campos *Integer*, *Float*, *Memo*, *Date* e *Time*. A **Listagem 12** contém a criação de todos os tipos de campos.

Tais métodos são bastante auto-explicativos. Você poderá notar que a maioria dos controles acaba utilizando o método *CreateStringControl*. Alguns utilizam máscaras. Em uma outra implementação, por exemplo, utilizei alguns componentes próprios para a manipulação de datas e horas. Existem vários disponíveis na internet. E você poderia substituí-los facilmente simplesmente mudando seus métodos *CreateControl*.

Listagem 11. Código para adição de controles em tela

```
procedure TfmAutoForm.AddControl(const Control: TControl;
   const DisplayLabel: string);
 const
   BETWEEN_CONTROL_AND_LABEL = 3;
   BETWEEN_CONTROLS_Y = 8;
   BETWEEN_CONTROLS_X = 10;
 var
   ALabel: TLabel;
   ATempMax: Integer;
 begin
   ALabel := TLabel.Create(Self);
   ALabel.Caption := DisplayLabel;
   if Control.Width < ALabel.Width then
     Control.Width := ALabel.Width;
   if (FCurWidth + Control.Width + MARGIN) > pnlControls.Width then
   begin
     FCurHeight := FMaxHeight - MARGIN + BETWEEN_CONTROLS_Y;
     FCurWidth := MARGIN;
   end;
   ALabel.Parent := pnlControls;
   ALabel.Left := FCurWidth;
   ALabel.Top := FCurHeight;
   Control.Parent := pnlControls;
   Control.Left := FCurWidth;
   if DisplayLabel = '' then
   begin
     Control.Top := FCurHeight;
     ALabel.Height := 0;
   end
   else
     Control.Top := FCurHeight + ALabel.Height + BETWEEN_CONTROL_AND_LABEL;
   ATempMax := Control.Top + Control.Height + MARGIN;
   if ATempMax > FMaxHeight then
     FMaxHeight := ATempMax;
   Inc(FCurWidth, Control.Width + BETWEEN_CONTROLS_X);
 end;
```

Testando a ferramenta

Vamos finalmente fazer um primeiro teste do que estamos desenvolvendo. Crie uma nova *unit* chamada *AutoFormFactory.pas*. Declare *AutoForm* e *Forms* na seção *uses* e declare uma classe chamada *TAutoFormFactory* como na **Listagem 13.**

Pressione *Ctrl* + *Shift* + *C* para que o Delphi crie o cabeçalho das funções. A **Listagem 14** possui o código da cada *procedure* da Classe.

Veja que criamos três procedures *CreateAndShowCadMarca*, *CreateAndShowCadVendedor*, *CreateAndShowCadCliente* mais o procedimento padrão para visualização do formulário criado. Veja que em todas as *procedures* chamamos o método *Create* da classe *TfmAutoForm*, chamamos o método adição do campo em tela de acordo com o tipo de campo e por fim fazemos uma chamada ao método *ShowForm* passando como parâmetro o formulário criado.

Em resumo: estamos apenas criando uma classe específica para nossa aplicação (já que a *unit TfmAutoForm* é bastante genérica) e, rapidamente, criamos três telas de cadastro bem simples.

Agora crie outro formulário que deverá ser o principal de sua aplicação e o único a ser criado automaticamente (configure isso em *Project>Options*). Salve-o como *Main.pas* e adicione botões para os cadastros (você pode se basear na **Figura 3**).

Listagem 12. Funções para criação dos campos em tela

```
procedure TfmAutoForm.AddIntegerField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer);
begin
  InternalAddIntegerField(FieldName, DisplayFieldName,
    DisplayWidth, sqAutoForm);
  InternalAddIntegerField(FieldName, DisplayFieldName,
    DisplayWidth, cdsAutoForm);
end;

procedure TfmAutoForm.AddFloatField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer);
begin
  InternalAddFloatField(FieldName, DisplayFieldName,
    DisplayWidth, sqAutoForm);
  InternalAddFloatField(FieldName, DisplayFieldName,
    DisplayWidth, cdsAutoForm);
end;

procedure TfmAutoForm.AddMemoField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer);
begin
  InternalAddMemoField(FieldName, DisplayFieldName,
    DisplayWidth, sqAutoForm);
  InternalAddMemoField(FieldName, DisplayFieldName,
    DisplayWidth, cdsAutoForm);
end;

procedure TfmAutoForm.AddDateField(const FieldName,
  DisplayFieldName: string);
begin
  InternalAddDateField(FieldName, DisplayFieldName, sqAutoForm);
  InternalAddDateField(FieldName, DisplayFieldName, cdsAutoForm);
end;

procedure TfmAutoForm.AddTimeField(const FieldName,
  DisplayFieldName: string);
begin
  InternalAddTimeField(FieldName, DisplayFieldName, sqAutoForm);
  InternalAddTimeField(FieldName, DisplayFieldName, cdsAutoForm);
end;

procedure TfmAutoForm.InternalAddIntegerField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer;
    const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
begin
  AddedKeyField := TIntegerField.Create(ToDataSet);
  AddedKeyField.FieldName := FieldName;
  AddedKeyField.DisplayLabel := DisplayFieldName;
  AddedKeyField.DisplayWidth := DisplayWidth;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;

  if ToDataSet = cdsAutoForm then
    CreateIntegerControl(AddedKeyField);
end;

function TfmAutoForm.CreateIntegerControl(const Field: TField):
TControl;
begin
  Result := CreateStringControl(Field);
end;

procedure TfmAutoForm.InternalAddFloatField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer;
    const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
begin
  AddedKeyField := TFloatField.Create(ToDataSet);
  AddedKeyField.FieldName := FieldName;
  AddedKeyField.DisplayLabel := DisplayFieldName;
  AddedKeyField.DisplayWidth := DisplayWidth;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;

  if ToDataSet = cdsAutoForm then
    CreateFloatControl(AddedKeyField);
end;

function TfmAutoForm.CreateFloatControl(const Field: TField):
TControl;
begin
  Result := CreateStringControl(Field);
end;

procedure TfmAutoForm.InternalAddMemoField(const FieldName,
  DisplayFieldName: string; DisplayWidth: Integer;
  const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
begin
  AddedKeyField := TStringField.Create(ToDataSet);
  AddedKeyField.FieldName := FieldName;
  AddedKeyField.DisplayLabel := DisplayFieldName;
  AddedKeyField.DisplayWidth := DisplayWidth;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;

  if ToDataSet = cdsAutoForm then
    CreateMemoControl(AddedKeyField);
end;

function TfmAutoForm.CreateMemoControl(const Field: TField):
TControl;
var
  AControl: TDBMemo;
begin
  AControl := TDBMemo.Create(Self);
  AControl.Width := pnlControls.Width - (MARGIN * 2);
  AControl.Height := 50;
  AControl.DataSource := dsAutoForm;
  AControl.DataField := Field.FieldName;
  AControl.ScrollBars := ssVertical;
  AddControl(AControl, Field.DisplayLabel);
  Result := AControl;
end;

procedure TfmAutoForm.InternalAddDateField(const FieldName,
  DisplayFieldName: string; const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
begin
  AddedKeyField := TDateField.Create(ToDataSet);
  AddedKeyField.FieldName := FieldName;
  AddedKeyField.DisplayLabel := DisplayFieldName;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;

  if ToDataSet = cdsAutoForm then
    CreateDateControl(AddedKeyField);
end;

function TfmAutoForm.CreateDateControl(const Field: TField):
TControl;
begin
  Field.EditMask := '!99/99/9999;1;_';
  Result := CreateStringControl(Field);
end;

procedure TfmAutoForm.InternalAddTimeField(const FieldName,
  DisplayFieldName: string; const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
begin
  AddedKeyField := TTimeField.Create(ToDataSet);
  AddedKeyField.FieldName := FieldName;
  AddedKeyField.DisplayLabel := DisplayFieldName;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;

  if ToDataSet = cdsAutoForm then
    CreateTimeControl(AddedKeyField);
end;

function TfmAutoForm.CreateTimeControl(const Field: TField):
TControl;
begin
  Field.EditMask := '!99:99;1;_';
  Result := CreateStringControl(Field);
end;
```

Listagem 13. Código da classe TAutoFormFactory

```
  TAutoFormFactory = class
  private
    class procedure ShowForm(const Form: TfmAutoForm);
  public
    class procedure CreateAndShowCadMarca;
    class procedure CreateAndShowCadCliente;
    class procedure CreateAndShowCadVendedor;
  end;
```

Listagem 14. Código das procedures da classe TAutoFormFactory

```
class procedure TAutoFormFactory.CreateAndShowCadMarca;
var
  AForm: TfmAutoForm;
begin
  AForm := TfmAutoForm.Create(Application, 'MARCA', 'Marcas', 'ID_MARCA', '');
  AForm.AddStringField('NOME', 'Nome', 20);
  AForm.AddStringField('CONTATO', 'Contato', 20);
  ShowForm(AForm);
end;
class procedure TAutoFormFactory.CreateAndShowCadVendedor;
var
  AForm: TfmAutoForm;
begin
  AForm := TfmAutoForm.Create(Application, 'VENDEDOR', 'Vendedor', 'ID_VENDEDOR', '');
  AForm.AddStringField('NOME', 'Nome', 20);
  AForm.AddIntegerField('MATRICULA', 'Matrícula', 10);
  ShowForm(AForm);
end;
class procedure TAutoFormFactory.CreateAndShowCadCliente;
var
  AForm: TfmAutoForm;
begin
  AForm := TfmAutoForm.Create(Application, 'CLIENTE', 'Cliente', 'ID_CLIENTE', '');
  AForm.AddStringField('NOME', 'Nome', 20);
  AForm.AddIntegerField('IDADE', 'Idade', 2);
  AForm.AddStringField('SEXO', 'Sexo', 1);
  ShowForm(AForm);
end;
class procedure TAutoFormFactory.ShowForm(const Form: TfmAutoForm);
begin
  Form.ShowModal;
  Form.Release;
  Form.Free;
end;
```

Listagem 15. Código para adição de um campo LookUp

```
procedure TfmAutoForm.AddLookupField(const FieldName, DisplayFieldName,
  KeyField, ListField, SQL: string; DisplayWidth: Integer);
begin
  InternalAddLookupField(FieldName, DisplayFieldName, KeyField, ListField, SQL,
    DisplayWidth, sqAutoForm);
  InternalAddLookupField(FieldName, DisplayFieldName, KeyField, ListField, SQL,
    DisplayWidth, cdsAutoForm);
end;
```

Listagem 16. Código de implementacao do campo LookUp

```
procedure TfmAutoForm.InternalAddLookupField(const FieldName, DisplayFieldName,
  KeyField, ListField, SQL: string; DisplayWidth: Integer;
  const ToDataSet: TDataSet);
var
  AddedKeyField: TField;
  AControl: TDBLookupComboBox;
begin
  AddedKeyField := TIntegerField.Create(ToDataSet);
  AddedKeyField.Visible := False;
  AddedKeyField.FieldName := FieldName;
  AddedKeyField.DisplayLabel := DisplayFieldName;
  AddedKeyField.DisplayWidth := DisplayWidth;
  SetUpWithReadMetaData(AddedKeyField);
  AddedKeyField.DataSet := ToDataSet;
  if ToDataSet = cdsAutoForm then
  begin
    AControl :=
      (CreateLookupControl(AddedKeyField, KeyField, ListField, SQL) as TDBLookupComboBox);
    AddedKeyField := TStringField.Create(ToDataSet);
    AddedKeyField.FieldKind := fkLookup;
    AddedKeyField.FieldName := 'LOOKUP' + ListField;
    AddedKeyField.DisplayLabel := DisplayFieldName;
    AddedKeyField.DisplayWidth := DisplayWidth;
    AddedKeyField.Size := AControl.ListSource.DataSet.FieldByName(ListField).Size;
    AddedKeyField.LookupDataSet := AControl.ListSource.DataSet;
    AddedKeyField.LookupKeyFields := KeyField;
    AddedKeyField.LookupResultField := ListField;
    AddedKeyField.KeyFields := FieldName;
    AddedKeyField.Lookup := True;
    AddedKeyField.DataSet := ToDataSet;
  end;
end;
```

Adicione *AutoFormFactory* à seção *uses* e implemente os manipuladores dos eventos dos três primeiros botões chamando os métodos da *Factory* (exemplo: *TAutoFormFactory.CreateAndShowCadVendedor*).

Agora, rode sua aplicação e abra seus formulários criados automaticamente. Teste tudo e comprove que um pouco de trabalho agora lhe poupará muito tempo de implementação.

Aperfeiçoando a criação dinâmica

Com isso, já podemos fazer bastante coisa, mas sabemos que ainda podemos melhorar e muito. Infelizmente, o espaço de um artigo é curto para podermos avançar até um ponto completamente satisfatório. Mas, ainda há algumas partes bastante interessantes que vale a pena estudarmos.

Vejamos a criação de um campo com suporte a *lookup* em outras tabelas. Adicione o seguinte método à seção *public* e implemente-o (**Listagem 15**). E como de costume implemente também seu método correspondente (**Listagem 16**).

A implementação desse método também é bastante auto-explicativa. A sua única peculiaridade está no fato da criação de dois campos. Primeiro, o campo onde será realmente armazenado o valor. Depois disso, o campo que irá mostrar o valor. Este segundo campo só precisará ser adicionado ao *TClientDataSet*, e não ao *TSQLQuery*, pois, de maneira alguma um valor será atualizado nesse. O *TDataSet* utilizado para que o *lookup* funcione é criado no método *CreateLookupControl*. Veja sua implementação na **Listagem 17**.

Como podemos ver, esse método cria um *TDataSource* e um *TSimpleDataSet*

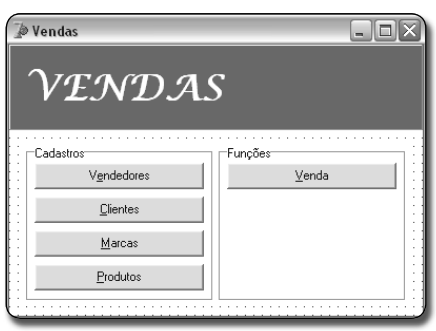


Figura 3. Configuração dos componentes do formulário principal

para servir de lista para o *lookup*. Um componente *TDBLookupComboBox* é então criado e configurado corretamente.

Agora, volte ao *AutoFormFactory* e crie um método para criação do cadastro de produtos como na **Listagem 18**. A *procedure* segue o mesmo padrão utilizado anteriormente para criação dos demais cadastros.

Faça também a chamada desse método no botão da janela principal e rode o programa. Você verá que o campo *Marca* aparecerá corretamente como *lookup*. Caso ainda não tenha feito, cadastre algumas marcas e teste o cadastro de produtos.

Os detalhes finais

Existem diversas outras funcionalidades interessantes que você ainda pode implementar. Vamos ver mais algumas idéias. Comecemos por alguns detalhes na abertura do formulário. Manipule o evento *OnShow* do formulário principal como na **Listagem 19**.

A primeira linha deste código configura o tamanho do painel onde estão os controles para não deixar muito espaço abaixo. Esteticamente, o painel abre com uma melhor aparência. Já o segundo trecho apenas varre a lista de controles em busca de um *TWinControl*, para torná-lo ativo (com foco). Se desejar, você pode copiar esta parte para o evento *AfterInsert* do *TClientDataSet*, para que, a cada vez que um registro for inserido, o foco volte ao primeiro controle.

Você também pode adicionar uma propriedade para configurar um índice padrão para o formulário criado:

```
property IndexFieldNames: string read
GetIndexFieldNames write SetIndexFieldNames;
```

Os métodos de leitura e escrita devem apenas ler e configurar *cdsAutoForm.IndexFieldNames*, respectivamente. Experimente testar essa nova propriedade modificando, por exemplo, o cadastro de clientes (em *AutoFormFactory*), configurando-a como *'NOME'*. Para finalizar, poderíamos atribuir valores *Default* aos campos. Uma das maneiras seria a seguinte:

Declare o seguinte *record* acima da declaração de *TfmAutoForm*:

```
TDefaultValue = record
  FieldName: string;
  DefaultValue: Variant;
end;
```

Declare também na seção *private* de seu formulário o *array* dinâmico:

```
FDefaultValues: array of TDefaultValue;
```

Agora, implemente o método (seção *public*) que permitirá a configuração de um valor *default*. Observe a **Listagem 20**.

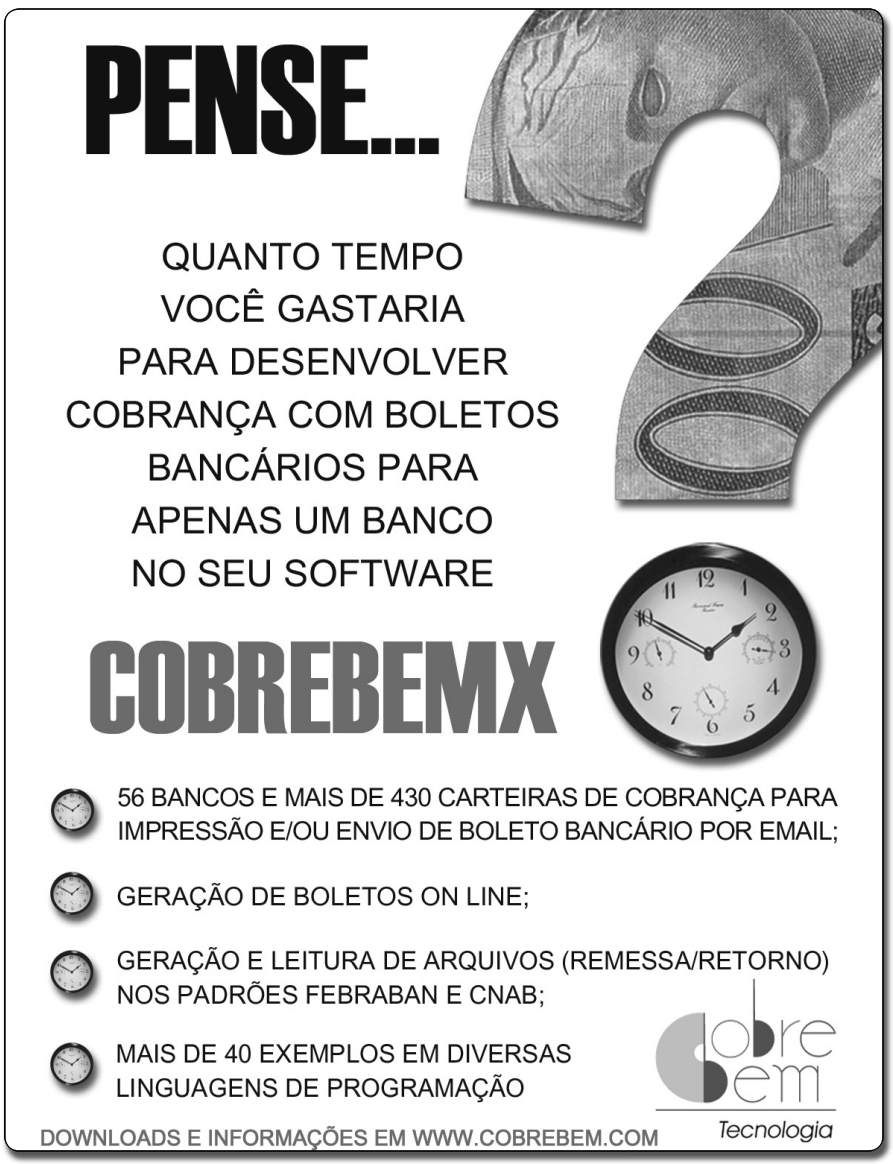
Desta maneira, como *FDefaultValues* é um *array* dinâmico de *TDefaultValues*, simplesmente adicionamos uma nova posição e configuramos o nome do campo e seu valor. Note que esse valor é do tipo *variant* para poder receber diversos tipos de dados. Finalizando, manipule o evento *OnNewRecord* de *cdsAutoForm* como na **Listagem 21**.

O primeiro *loop* varre os campos do *TClientDataSet* e, caso encontre algum do tipo *TDateTime*, configura-o com a data atual. Já o segundo *loop* varre o *array* de valores *default* e configura-os nos campos correspondentes.

Teste essa nova implementação chamando *SetDefaultValue* passando campo e valor quaisquer.

Ainda existem outras coisas muito práticas a serem feitas para facilitar nosso dia-a-dia. Outros tipos de campo também podem ser adicionados, como uma lista *ComboBox* ou o tipo *Boolean*. No entanto, esta tarefa ficará para o leitor.

Já a criação da tela de vendas propriamente dita foge um pouco ao con-



Listagem 17. Código para adição de um campo LookUp

```
function TfmAutoForm.CreateLookupControl(const Field: TField;
  const KeyField, ListField, SQL: string): TControl;
var
  AListSource: TDataSource;
  ADataSet: TSimpleDataSet;
  AControl: TDBLookupComboBox;
begin
  AListSource := TDataSource.Create(Self);
  ADataSet := TSimpleDataSet.Create(Self);
  ADataSet.Connection := conAutoForm;
  ADataSet.DataSet.CommandText := SQL;
  ADataSet.Open;
  AListSource.DataSet := ADataSet;
  AControl := TDBLookupComboBox.Create(Self);
  AControl.Parent := pnlControls;
  AControl.Width := Field.DisplayWidth * 6;
  AControl.KeyField := KeyField;
  AControl.ListField := ListField;
  AControl.ListSource := AListSource;
  AControl.DataSource := dsAutoForm;
  AControl.DataField := Field.FieldName;
  AddControl(AControl, Field.DisplayLabel);
  Result := AControl;
end;
```

Listagem 18. Procedure de criação do formulário de cadastro de Produtos

```
class procedure TAutoFormFactory.CreateAndShowCadProduto;
var
  AForm: TfmAutoForm;
begin
  AForm := TfmAutoForm.Create(Application, 'PRODUTO', 'Produto', 'ID_PRODUTO', '');
  AForm.AddStringField('NOME', 'Nome', 20);
  AForm.AddLookupField('ID_MARCA', 'Marca', 'ID_MARCA', 'NOME',
    'select ID_MARCA, NOME from MARCA', 20);
  AForm.AddFloatField('PRECO', 'Preço', 10);
  AForm.AddFloatField('QUANTIDADE_ESTOQUE', 'Quantidade em Estoque', 10);
  ShowForm(AForm);
end;
```

Listagem 19. Detalhes na abertura do formulário principal

```
procedure TfmAutoForm.FormShow(Sender: TObject);
var
  i: Integer;
begin
  inherited;
  pnlControls.Height := FMaxHeight;
  for i := 0 to pnlControls.ControlCount - 1 do
  begin
    if pnlControls.Controls[i] is TWinControl then
    begin
      ActiveControl := (pnlControls.Controls[i] as TWinControl);
      Break;
    end;
  end;
end;
```

Listagem 20. Método para configuração de valores default

```
procedure TfmAutoForm.SetDefaultValue(const FieldName: string;
  const Value: Variant);
begin
  SetLength(FDefaultValues, Length(FDefaultValues) + 1);
  FDefaultValues[High(FDefaultValues)].FieldName := FieldName;
  FDefaultValues[High(FDefaultValues)].DefaultValue := Value;
end;
```

Listagem 21. Evento NewRecord do ClientDataSet

```
procedure TfmAutoForm.cdsAutoFormNewRecord(DataSet: TDataSet);
var
  i: Integer;
begin
  for i := 0 to cdsAutoForm.FieldCount - 1 do
  begin
    if cdsAutoForm.Fields[i] is TDateTimeField then
      cdsAutoForm.Fields[i].AsDateTime := Now;
  end;

  for i := 0 to High(FDefaultValues) do
  begin
    cdsAutoForm.FieldByName(FDefaultValues[i].FieldName).Value :=
      FDefaultValues[i].DefaultValue;
  end;
end;
```

texto do artigo, portanto, resolvemos aproveitar melhor este espaço com o aperfeiçoamento da técnica apresentada. Deste modo, tal tarefa também fica como treino.

Conclusão

Quando trabalhamos em grandes projetos, rapidamente aprendemos que o tempo gasto na automação de tarefas cotidianas resulta em posterior agilidade. Isso reduz a quantidade de trabalho mecânico e lento e nos faz focar no que realmente interessa, as regras de negócio. Deste pensamento nasceram muitos conceitos da engenharia de *software*, como componentização, *frameworks*, *software factories*, *generic programming*, *language oriented programming*, dentre outros.

Além de estudarmos um pouco da automação da criação de formulários, também pudemos observar nesse artigo alguns outros conceitos e técnicas de programação *Delphi* muito importantes e interessantes.

Pratique bastante e tente introduzir tais técnicas em seus projetos. Certamente, você e sua equipe só terão a ganhar. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

10 Motivos para hospedar seu site na RedeHost!

30 dias GRÁTIS para testar.

Mais de 15 mil clientes e 6 anos no mercado.

Até 50Gb de espaço para e-mails ou site.

Planos a partir de R$ 13,90.

Webmail em Ajax com calendário, tarefas, anotações e RSS.

ASP, PHP, .net e MySQL no mesmo plano.

Domínios adicionais ilimitados (apontam para o site principal).

Data center no Brasil com Links Redundantes.

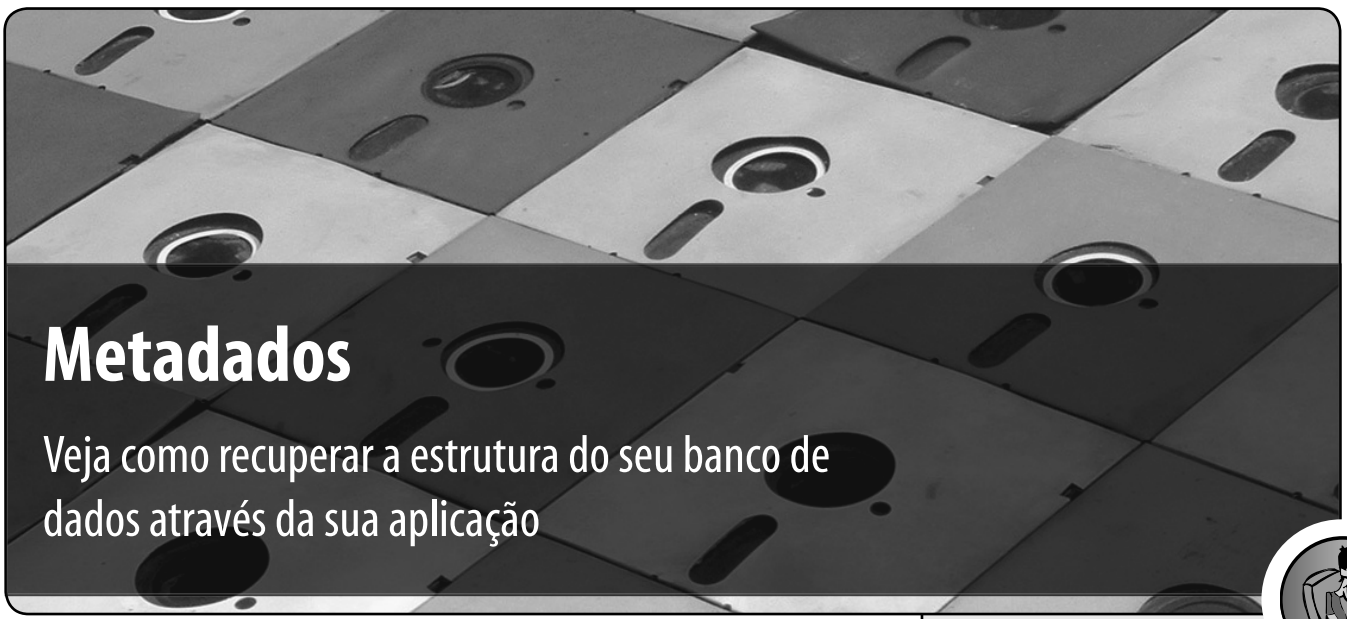
Infra-estrutura com servidores Dell e redes Cisco.

Estatísticas de acesso com 75 relatórios.

RedeHost

www.redehost.com.br

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET



Metadados

Veja como recuperar a estrutura do seu banco de dados através da sua aplicação

Resumo DevMan

Os MetaDados são informações valiosíssimas em determinados momentos de nossa aplicação. Podemos utilizar os metadados para recuperar informações sobre o BD, como tabelas, campos, índices, procedures, triggers e outros objetos.

Nesse artigo veremos

• O que são MetaDados?

• Acesso aos MetaDados do banco.

Qual a finalidade

• A finalidade dos metadados é conhecer a estrutura do banco de dados que está se trabalhando e tirar proveito dessas informações.

Quais situações utilizam esses recursos?

• Aplica-se a todo tipo de software que se utiliza de banco de dados Win32.

Maikel Marcelo Scheid

maikelscheid@gmail.com

é técnico em Informática com ênfase em Análise e Programação de Sistemas. Atua na área de Desenvolvimento de Softwares em Delphi para plataforma Win32 e .NET com banco de dados Firebird e MS SQL. É membro da equipe Editorial ClubeDelphi.

Chamados também *Metainformação,* os *Metadados* são informações referentes a uma base de dados (descritas na maior parte das literaturas como *dados sobre dados),* sendo definidas também como abstração dos dados que indicam a estrutura e características de uma fonte de armazenamento de valores (tabelas, atributos, procedimentos, *triggers* etc). É a partir de suas informações que ocorre o processamento e funcionamento da estrutura de armazenagem, pesquisa e manutenção das informações.

Sabemos que no mercado há uma série de ótimas ferramentas *case* para a manipulação e gerenciamento destas informações e que há também casos em que estas informações precisam ser tratadas a partir de um sistema, que deverá acessar e manipular os resultados obtidos fazendo assim uma análise crítica da estrutura da fonte de armazenamento, levando o programador a tomar decisões.

Uma das maiores finalidades ao extrairmos os metadados em nossas aplicações refere-se a alguma mudança de estrutura das tabelas do banco de dados, criação de alguma nova tabela, campo, chave, entre outras situações nas quais o programador

verifica através dos metadados da estrutura atual se houve ou não a atualização, podendo passar parâmetros definidos no caso de base desatualizada.

Faremos no decorrer deste artigo o acesso aos metadados do banco de dados *Employee.fdb* que acompanha a instalação *default* do banco de dados *Firebird*, onde veremos como recuperar informações da estrutura de modo geral, tabelas, atributos, relacionamentos, campos chaves entre outros, que também utilizaremos para a montagem de um *Builder SQL* que irá gerar automaticamente alguns comandos de relacionamento de acordo com a atual estrutura da fonte (*select* e *update* onde o usuário poderá selecionar quais os campos deverão ser adicionados à consulta).

Criando a aplicação

Utilizaremos no decorrer deste artigo para criação do nosso exemplo a versão 7 do *Delphi*, onde criaremos uma estrutura que irá se conectar através dos componentes da paleta *dbExpress* à uma base de dados *Firebird 1.5*, mas que poderá ser utilizada outra versão do Delphi e/ou outra versão do *Firebird*. Criaremos um exemplo que depois de conectado à base, irá exibir a lista de tabelas do banco de dados, tabelas geradas automaticamente pelo sistema (*System*), procedimentos e demais informações.

No menu *File|New>Application* crie um novo projeto *Win32* e salve a *Unit* e o projeto em um diretório como *uMetadados.pas* e *prjMetadados.dpr*, respectivamente. Altere a propriedade *Name* do formulário principal para *FrmMetadados* e o *Caption* para *Extração de Metadados*.

Nosso próximo passo será a configuração da conexão com a base de dados *Employee.fdb*. Adicione um componente *SQLConnection*(*Conexao*) da paleta *dbExpress*, que será o responsável pela conexão com o arquivo físico do banco de dados, que deixaremos de forma dinâmica, a fim de você poder extrair os metadados de qualquer arquivo de dados. Ainda no componente *Conexao*, defina sua propriedade *LoginPrompt* para *False* fazendo com que a senha não seja solicitada em conexões posteriormente.

As demais configurações de conexão serão feitas em *runtime* antes de abrirmos a conexão, onde o endereço físico da fonte de dados também será informado ao componente. Para a configuração adicione no formulário um componente *Label* com o *Caption Localizar fonte de dados Firebird* e logo abaixo um *Edit* (*EdtCaminho)* onde iremos exibir o arquivo de conexão selecionado, adicionando ao lado deste um componente *SpeedButton* (*btnBuscar*) da paleta *Additional*, definindo sua propriedade *Caption* para "..." e *Flat* para *True*, componente este que utilizaremos para realizar a busca dos arquivos de banco de dados **.fdb*. Para realizarmos a busca destes arquivos, adicione também da paleta *Dialogs* um componente *OpenDialog* (*OpenCaminho*) e configure na sua propriedade *Filter* de modo que na primeira linha da lista *Filter name* seja preenchido o valor *Base de dados Firebird* (**.fdb*) e na coluna ao lado, na lista *Filter* adicione o filtro de busca **.fdb* (**Figura 1**), que irá exibir apenas arquivos que se enquadram nesta extensão.

Para que o filtro e a pesquisa sejam ativados, adicione ao evento *OnClick* do *btnBuscar* o código a seguir, que será responsável pela chamada da caixa de diálogo para procurar os arquivos e depois de selecionado irá repassar o endereço do arquivo antecedido pelo endereço *IP* do *host* (*localhost*) ao componente *edtCaminho*.

```
if OpenCaminho.Execute then
begin
  EdtCaminho.Text :=
    '127.0.0.1:' + OpenCaminho.FileName;
end;
```

Logo abaixo do *EdtCaminho* iremos adicionar mais quatro componentes *Label* (a propriedade *Caption* deverá ser alterada da seguinte forma: *Usuário, Senha, Charset* e *Dialecto)* e quatro componentes *Edit* (*edtUsuario, edtSenha, edtCharset, edtDialecto*). No componente *edtSenha*, altere a propriedade *PasswordChar* para o caractere *, o que fará que ao digitarmos a senha nenhum caracter verdadeiro será exibido, aparecendo apenas o símbolo * para cada caractere preenchido. A propriedade *Text* destes componentes recém adicionados poderá ser preenchida com valores *default*, que são comuns à instalação padrão do *Firebird* utilizados na conexão com uma base de dados. No *edtUsuario* informe *Text* como *SYSDBA*, no *edtSenha* digite o valor *masterkey*, porém irão aparecer somente os caracteres *********, para o *edtCharset* atribua o valor *NONE* e em *edtDialecto* utilizaremos o valor *3*.

Ao lado do *edtCharset*, adicione ainda um *Button* (*btnConectar*) da paleta *Standard* e altere o *Caption* do mesmo para *CONECTAR*. Este componente será o responsável por passar os parâmetros digitados na tela ao componente de conexão e concretizar a mesma.

Antes de codificarmos o nosso projeto, precisaremos ainda adicionar ao formu-

Nota do DevMan

Ao utilizar ferramentas de front-end para bandos de dados, como o IBExpert, por exemplo, podemos extrair facilmente os metadados de bancos de dados. Esses metadados nada mais são do que a estrutura, o esqueleto do banco de dados, o esquema (*schema*). Com ele podemos recriar facilmente um BD sem a real necessidade em se conhecer a fundo o banco.

Veja como extrair os metadados diretamente no IBExpert. Entre no programa e conecte-se a uma base de dados Interbase ou Firebird. Em seguida acesse o menu *Tools>Extract Metadata*. Uma janela será aberta e nela você deverá selecionar de quais objetos quer extrair os metadados. Selecione *Extract all* para extrair a estrutura inteira do banco. Agora clique na aba *Data Tables*. É possível agora selecionar todas as tabelas do seu banco de dados para extrair os registros, para essa aplicação não há necessidade de escolher uma tabela em específico. Simplesmente clique no botão onde o ícone são duas setas para a direita. Por fim, escolha as opções e selecione um caminho para o banco. Agora clique em *Start Extract* e aguarde. Pronto, seus metadados estão extraídos e agora com base nisso podemos criar um novo banco.

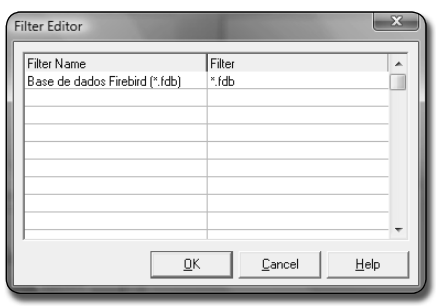


Figura 1. Configurando propriedade Filter para pesquisa

lário três componentes *Label* (alterando a propriedade *Caption* respectivamente para *Tabelas do banco de dados, Tabelas do Sistema no BD* e *Procedures)* e logo abaixo de cada *Label*, adicionar um componente *ListBox* (*ListaTabelas, ListaTabelasSistema* e *ListaProcedimentos),* conforme estrutura organizada na **Figura 2**.

Nos *ListBoxes* adicionados, exibiremos algumas das informações de *Metadados* que obteremos da base conectada, na primeira listagem todas as tabelas criadas para armazenamento de registros na base, a listagem seguinte exibirá as tabelas criadas pelo sistema, tabelas estas que são utilizadas para controle interno e manutenção do próprio banco de dados e por último, a lista contendo todos os procedimentos existentes na base conectada.

Para realizar a conexão à base de dados, adicione ao evento *OnClilck* do *btnConectar* o código da **Listagem 1** que encontra-se comentado para um entendimento mais fácil, onde primeiramente estamos realizando uma validação do preenchimento de todas as informações. Logo após a passagem destas informações aos parâmetros necessários pelo componente *Conexao* à base de dados e abertura do mesmo, já são extraídas algumas informações de *Metadados* e exibidas na tela. Caso a primeira condição não seja atendida ou o endereço da base de dados não foi informado, chamamos o diálogo de conexão e solicitaremos ao usuário que selecione um novo arquivo para extração dos metadados.

Neste momento, ao executarmos a aplicação já será possível a extração de algumas informações da base conectada.

Listagem 1. Conectando ao arquivo da base de dados

```
procedure TFrmMetadados.btnConectarClick(Sender: TObject);
begin
  { Validação dos campos preenchidos }
  if EdtCaminho.Text <> '' then
  begin
    if ((EdtUsuario.Text <> '') and (edtSenha.Text <> '') and
      (edtCharset.Text <> '') and (EdtDialecto.Text <> '')) then
    begin
      { Passagem dos parâmetros para o componente de conexão }
      CONEXAO.Params.Values['DataBase']      := EdtCaminho.Text;
      CONEXAO.Params.Values['User_Name']     := EdtUsuario.Text;
      CONEXAO.Params.Values['Password']      := edtSenha.Text;
      CONEXAO.Params.Values['CHARACTERSET']  := edtCharset.Text;
      CONEXAO.Params.Values['SQLDialect']    := EdtDialecto.Text;
      { Abertura do Componente de Conexão }
      CONEXAO.Open;
      { Obtém o nome das tabelas de armazenamento da base }
      CONEXAO.GetTableNames(ListaTabelas.Items);
      { Obtém o nome das tabelas de controle do sistema na base }
      CONEXAO.GetTableNames(ListaTabelasSistema.Items,True);
      { Obtém a lista de procedimentos da base }
      CONEXAO.GetProcedureNames(ListaProcedimentos.Items);
    end
    else
    MessageDlg('Preencha os demais campos necessários para a
conexão!',mtWarning,[mbOK],0);
  end
  else
  begin
    { Caso nenhum arquivo foi selecionando, a seleção ocorre neste momento }
    MessageDlg('Selecione um arquivo de base de dados', mtWarning,[mbOK],0);
    if OpenCaminho.Execute then
      EdtCaminho.Text := '127.0.0.1:' + OpenCaminho.FileName;
    if EdtCaminho.Text <> '' then
      btnConectar.Click;
  end;
  { Exibe mensagem quando conexão bem sucedida }
  if CONEXAO.Connected then
    MessageDlg('Conectado ao arquivo da fonte de dados!',
    mtInformation,[mbOK],0);
end;
```

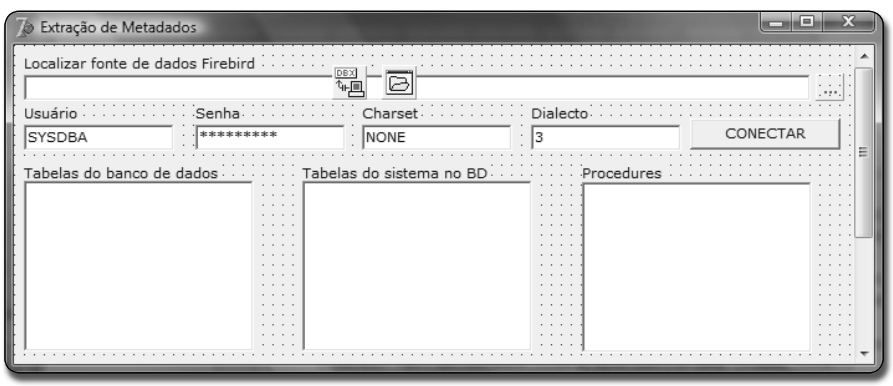


Figura 2. Organização de estrutura do sistema

Obtendo os Fields de uma tabela

A obtenção dos *Fields* de uma tabela é feito através de um simples processo e pouca codificação. Adicione ao formulário, logo abaixo da lista das tabelas da base um novo componente *Label* e altere seu *Caption* para *Fields da Tabela* e adicione abaixo do mesmo um novo componente *ListBox* (*ListaFieldsTabela).* A exibição dos *Fields* será feita ao clicarmos sobre uma das tabelas obtidas no componente *ListaTabelas,* ao qual será necessário adicionar a seguinte linha de códigos ao seu evento *OnClick:*

```
CONEXAO.GetFieldNames(ListaTabelas.
Items.Strings[ListaTabelas.
ItemIndex],ListaFieldsTabela.Items);
```

O código fará a chamada do método *GetFieldNames* do componente *Conexao* e receberá como parâmetro o nome da tabela selecionada na lista e o local onde os *Fields* deverão ser exibidos. Neste processo, utilizando o componente *SQLConnection* não será possível obtermos os tipos e tamanho de cada *Field,* nem se os mesmos pertencem ou não aos campos chave da tabela selecionada, onde para este processo implantaremos uma nova listagem e obteremos os resultados a partir de uma consulta *SQL* ao banco de dados conectado.

Obtendo chaves, tipo e tamanho dos fields

Para exibir estas informações faremos a criação de um processo diferenciado ao que vimos até o momento para obtenção dos *metadados* da fonte de dados selecionada. Precisaremos adicionar ao formulário um *Button* (*btnCompleta*) alte-

rando seu *Caption* para *< Inf. Completas >*, onde depois mais na codificação iremos criar um objeto do tipo *SQLDataSet* em memória e configurá-lo com a instrução a ser pesquisada na base. Os resultados serão listados em um componente *ListView*(*LstvInfCompletas)* que deverá ser adicionado ao lado do *btnCompleta.*

Utilizaremos ainda para identificação dos campos chaves no *LstvInfCompletas* imagens inseridas em um componente *ImageList*(*ImgChaves*). Na configuração das imagens, dê um duplo clique sobre o componente *ImgChaves* e adicione duas imagens, sendo a imagem com *ID*=0 que irá identificar um campo chave (*Key field*) e a imagem com *ID*=1 a que irá identificar um campo normal da tabela (**Figura 3**).

Na configuração do componente *LstvInfCompletas*, algumas de suas propriedades deverão ser alteradas, iniciando pela propriedade *CheckBoxes* que deverá ser alterada para *True* fazendo com que cada item adicionado no componente possua um componente *CheckBox* para ser selecionado e utilizado posteriormente para criação de rotinas de códigos *SQL*. Na propriedade *SmallImages* do componente selecione o componente *ImgChaves* que será a origem das imagens de miniaturas que irão diferenciar campos comuns de campos chave. Por último, a propriedade *ViewStyle* deverá ser alterada para *vsSmallIcon*, fazendo com que os resultados

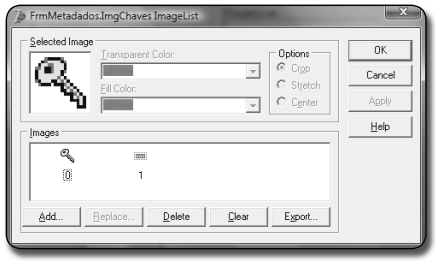


Figura 3. Adicionando imagens para identificação dos campos

sejam exibidos em uma lista contínua e contendo ícones.

Selecionando o botão *btnCompleta*, dê um duplo clique sobre o mesmo e adicione ao seu evento *OnClick* o código da **Listagem 2,** onde iremos criar uma *function* encapsulada dentro do procedimento *OnClick* do

Listagem 2. Exibindo informações completas dos Fields de uma tabela

```
procedure TFrmMetadados.btnCompletaClick(Sender: TObject);
  { Criaremos uma função para verificar se o campo é uma chave da
tabela }
  function CampoChave(FieldName, TableName: String): Integer;
  var
    { Objeto para pesquisar campo na base de dados }
    { Declare a Uses "SQLExpr" }
    SQLp : TSQLDataSet;
  begin
    try
      {Cria em runtime o componente de acesso e pesquisa ao BD}
      SQLp := TSQLDataSet.Create(Self);
      with SQLp do
      begin
        { Relaciona o componente de pesquisa à conexão ao BD }
        SQLConnection := CONEXAO;
        Close;
        { Passa instrução ao componente de pesquisa }
        CommandText := 'select i.rdb$field_name '+
        'from rdb$relation_constraints rc,
           rdb$index_segments i, rdb$indices idx '+
        'where (i.rdb$index_name = rc.rdb$index_name) and '+
        '(idx.rdb$index_name = rc.rdb$index_name) and '+
        '(rc.rdb$constraint_type = ''PRIMARY KEY'') '+
        'and (i.rdb$field_name = :FieldName) '+
        'and (rc.rdb$relation_name = :TableName) '+
        'order by rc.rdb$relation_name, i.rdb$field_position';
        { Passa nome do Field como parâmetro }
        Params[0].Value := FieldName;
        { Passa nome da Tabela como parâmetro }
        Params[1].Value := TableName;
        { Executa a pesquisa }
        Open;
        { se houve retorno o resultado será 0 = Imagem de Chave e se não
          houve retorno o resultado será 1 = Imagem de campo não chave}
        if not IsEmpty then
        Result := 0 else
        Result := 1;
      end;
    finally
      { Limpa o componente da memória }
      SQLp.Free;
    end;
  end;
  { Fim da função declarada e início do procedimento executado normalmente }
var
  { Declare a Uses "SQLExpr" }
  SQLp : TSQLDataSet;
  { Variáveis para criação da lista de fields }
  List:TStringList;
  I, DisplayIcon :Integer;
  DisplayName:String;
  Item:TListItem;
begin
  { Limpa os itens adicionados }
  LstvInfCompletas.Clear;
  { Cria o objeto para listar os fields }
  List := TStringList.Create;
  { Aloca um espaço na memória para adicionar ícones }
  Icon := TIcon.Create;
  { Cria em runtime o componente de acesso e pesquisa ao BD }
  SQLp := TSQLDataSet.Create(Self);
  with SQLp do
  begin
    { Relaciona o componente de pesquisa à conexão ao BD }
    SQLConnection := CONEXAO;
    { Fecha o componente para receber a instrução }
    Close;
    { Passagem da instrução ao componente }
    CommandText :=
    'SELECT RDB$RELATION_FIELDS.RDB$FIELD_NAME AS COLUMN_NAME, '+
    '  case '+
    '    when '+
    '      rdb$types.rdb$type_name = ''VARYING'' then ''VARCHAR'' '+
    '    when '+
    '      rdb$types.rdb$type_name = ''LONG'' then ''INTEGER'' '+
    '    else rdb$types.rdb$type_name '+
    '  end field_type, '+
    'rdb$fields.rdb$field_length field_size  '+
    'FROM RDB$RELATIONS '+
    'INNER JOIN RDB$RELATION_FIELDS ON
      RDB$RELATIONS.RDB$RELATION_NAME =
      RDB$RELATION_FIELDS.RDB$RELATION_NAME '+
    'LEFT JOIN RDB$FIELDS ON RDB$RELATION_FIELDS.RDB$FIELD_SOURCE =
      RDB$FIELDS.RDB$FIELD_NAME '+'LEFT JOIN RDB$TYPES ON
      RDB$FIELDS.RDB$FIELD_TYPE = RDB$TYPES.RDB$TYPE '+
    'WHERE RDB$RELATIONS.RDB$RELATION_NAME = :TabelName '+
    'AND RDB$RELATIONS.RDB$SYSTEM_FLAG = 0 '+
    'AND RDB$TYPES.RDB$FIELD_NAME = ''RDB$FIELD_TYPE'' '+
    'ORDER BY RDB$RELATION_FIELDS.RDB$FIELD_POSITION';
    { Passagem do nome da tabela selecionada ao parâmetro }
    Params[0].Value := ListaTabelas.Items.Strings[ListaTabelas.ItemIndex];
    { Abertura da pesquisa }
    Open;
  end;
  { Posiciona pesquisa no primeiro registro }
  SQLp.First;
  { Percorre todos os registros obtidos }
  while not SQLp.Eof do
  begin
    { Preenche nome do campo, tipo e tamanho }
    DisplayName := Trim(SQLp.Fields[0].AsString) + ' (' +
    Trim(SQLp.Fields[1].AsString) + ',' +
    SQLp.Fields[2].AsString + ')';
    { Chama novo procedimento para verificar se campo é uma chave }
    DisplayIcon := CampoChave(SQLp.Fields[0].
      AsString,ListaTabelas.Items.Strings[ListaTabelas.ItemIndex]);
    { Adiciona campo ao componente exibindo-o na tela }
    if DisplayName <> '' then
    begin
      Item := LstvInfCompletas.Items.Add;
      Item.Caption := DisplayName;
      Item.ImageIndex := DisplayIcon;
    end;
    { Avança ao próximo registro }
    SQLp.Next;
  end;
end;
```

botão. Ao declararmos uma função encapsulada não precisaremos declarar ela na seção de *Interface* da *uses*, mas poderemos utilizá-la somente dentro deste único procedimento. Nesta função definimos uma variável para pesquisa através de uma instrução *SQL* ao banco de dados.

Nessa instrução *SQL* verificamos se na tabela o campo recebido como parâmetro é um item chave ou não. Caso a instrução *SQL* retorne algum valor, significa que se trata de um *key field* e passamos a ele o valor 0, que será o *ID* da imagem que o mesmo deverá assumir. Campos não chaves (que não retornam valores) receberão o valor 1, *ID* da imagem de campos não chave.

Finalizada a função encapsulada, teremos o procedimento normal onde também teremos a declaração de variáveis que serão utilizadas para consulta dos *Fields* à base de dados e preenchimento da lista no formulário. Na instrução *SQL* aplicada, passamos como parâmetro a tabela da qual desejamos saber os *fields*, tipo e tamanho dos mesmos, que depois mais serão tratados um a um em um laço que define a descrição e tipo com a qual aparecerão no formulário.

Finalizado a codificação do botão, execute a aplicação e realize a conexão com uma base de dados *Firebird* de modo que já seja possível visualizar as tabelas disponíveis na base. Ao selecionar uma tabela, observe que todos os campos da mesma serão exibidos, porém sem informações de tipo e tamanho, disponíveis após o clique sobre o botão, para obter mais informações referentes aos *fields* da tabela selecionada (**Figura 4**).

Criando instruções SQL

Simularemos nesta etapa do artigo as funcionalidades de um *Query Builder*, onde a partir dos campos selecionados na lista montaremos instruções *SQL* de pesquisa (*select*) e alteração (*update*). Nossa referência será quanto aos itens marcados no componente *LstvInfCompletas*, para o qual será necessário adicionarmos dois botões (*btnSQLSelect* e *btnSQLUpdate)* alterando sua propriedade *Caption* respectivamente para *SQL Select* e *SQL Update*. Em ambos os botões faremos uma pesquisa dentre todos os itens selecionados existentes na lista e adicionaremos os mesmos à variável temporária que irá armazenar o comando a ser exibido. Utilizaremos uma variável auxiliar para controle dos sinais de separação dos campos tanto na pesquisa como também

Listagem 3. Codificação dos botões para montagem das instruções SQL

```
procedure TFrmMetadados.btnSQLSelectClick(Sender: TObject);
var
  i : integer;
  Anterior : integer;
  Select : TStringList;
  Separador : String;
begin
  { Criação de StringList em memória }
  Select := TStringList.Create;
  Anterior  := 0;
  Separador := '';
  { Inicia a montagem da instrução }
  Select.Append('SELECT ');
  { Verifica os itens selecionados }
  for i := 0 to LstvInfCompletas.Items.Count -1 do
  begin
    { Adiciona item selecionado }
    if LstvInfCompletas.Items.Item[i].Checked then
    begin
      { Adiciona caracter de separação dos campos }
      if Anterior > 0 then
      begin
        Select.Insert(Select.Count,Select.Strings[Select.Count-1]+', ');
        Select.Delete(Select.Count-2);
      end;
      { Adiciona campo e separador à lista }
      Select.Add(Separador + Copy(LstvInfCompletas.Items.Item[i].
      Caption,1,Pos(' ',LstvInfCompletas.Items.Item[i].Caption)));
      Anterior := 1;
    end;
  end;
  { Finaliza a instrução e passa nome da tabela }
  Select.Add('FROM '+ListaTabelas.Items.Strings[ListaTabelas.ItemIndex]);
  { Copia instrução para a memória }
  Clipboard.AsText := Select.Text;
  { Mensagem na tela com instrução }
  ShowMessage('Comando SQL gerado com sucesso!'+#13+#13+
   'Comando SQL Copiado para a memória.'+#13+#13+Select.Text);
end;
procedure TFrmMetadados.btnSQLUpdateClick(Sender: TObject);
var
  i : integer;
  Anterior : integer;
  Select : TStringList;
  Separador : String;
begin
  { Criação de StringList em memória }
  Select := TStringList.Create;
  Anterior  := 0;
  Separador := '';
  { Inicia a montagem da instrução }
  Select.Append('UPDATE ' + ListaTabelas.Items.
    Strings[ListaTabelas.ItemIndex]+ ' SET ');
  { Verifica os itens selecionados }
  for i := 0 to LstvInfCompletas.Items.Count -1 do
  begin
    { Adiciona item selecionado }
    if LstvInfCompletas.Items.Item[i].Checked then
    begin
      { Adiciona campo e separador à lista }
      if Anterior > 0 then
      begin
        Select.Insert(Select.Count,Select.Strings[Select.Count-1]+', ');
        Select.Delete(Select.Count-2);
      end;
      { Finaliza a instrução e aguarda parâmetros }
      Select.Add(Separador + Copy(LstvInfCompletas.Items.Item[i].
      Caption,1,Pos(' ',LstvInfCompletas.Items.Item[i].Caption))+' = :p'+
      Copy(LstvInfCompletas.Items.Item[i].Caption,1,Pos('
      ',LstvInfCompletas.Items.Item[i].Caption)));
      Anterior := 1;
    end;
  end;
  { Adiciona o Where para receber parâmetros }
  Select.Add(' WHERE ');
  Anterior := 0;
  { Percorre todos os registros e localiza registros chaves }
  for i := 0 to LstvInfCompletas.Items.Count -1 do
  begin
    { adiciona campo chave como parâmetro }
    if LstvInfCompletas.Items.Item[i].ImageIndex = 0 then
    begin
      { Adiciona concatenação dos parâmetros }
      if Anterior > 0 then
      begin
        Select.Insert(Select.Count,Select.Strings[Select.Count-1]+' and ');
        Select.Delete(Select.Count-2);
      end;
      { Finaliza passagem dos parâmetros e SQL }
      Select.Add(Separador + Copy(LstvInfCompletas.Items.Item[i].
      Caption,1,Pos(' ',LstvInfCompletas.Items.Item[i].Caption))+' = :p'+
      Copy(LstvInfCompletas.Items.Item[i].Caption,1,Pos('
      ',LstvInfCompletas.Items.Item[i].Caption)));
      Anterior := 1;
    end;
  end;
  { Copia SQL para a memória }
  Clipboard.AsText := Select.Text;
  { Mensagem na tela com instrução }
  ShowMessage('Comando SQL gerado com sucesso!'+#13+#13+'Comando
SQL Copiado para a memória.'+#13+#13+Select.Text);
end;
```

na passagem dos parâmetros. Após a passagem por todos os itens, finalizamos a instrução *SQL* adicionando-a à área de transferência para que seja colada em qualquer outro programa e exibimos uma mensagem de sucesso na tela.

Adicione ao evento *OnClick* de cada botão o código relacionado na **Listagem 3** que encontra-se comentado para facilitar o entendimento. Após a codificação de ambos os botões, execute a aplicação e observe as rotinas de *SQL* criadas, e faça o teste em qualquer outro gerenciador.

Concluído o processo de codificação dos botões, ao executarmos o sistema e obtermos as instruções *SQL* de *Select* (**Figura 5**) e a instrução de *Update* (**Figura 6**) observaremos que em ambos foram adicionados apenas os itens selecionados, observando corretamente a organização dos caracteres para separação dos campos e atribuição dos parâmetros na condição *Where* na instrução de *Update,* onde todos os campos do tipo *Key* foram adicionados.

Conclusão

Vimos neste artigo alguns métodos para extração dos *metadados* de arquivos de banco de dados. Utilizamos alguns métodos que serão úteis para você aplicar em situações onde necessita analisar a estrutura de um cliente, e/ou executar um procedimento para correção e atualização de banco de dados. Além das informações obtidas na demonstração deste artigo, você poderá criar rotinas que lhe tragam outros tipos de informações como índices, *triggers*, *generators* etc. Amplie também o *Query Builder* de forma a criar *inner joins* entre tabelas ou comandos para inserção e exclusão de registros. Abraço e até o próximo artigo. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

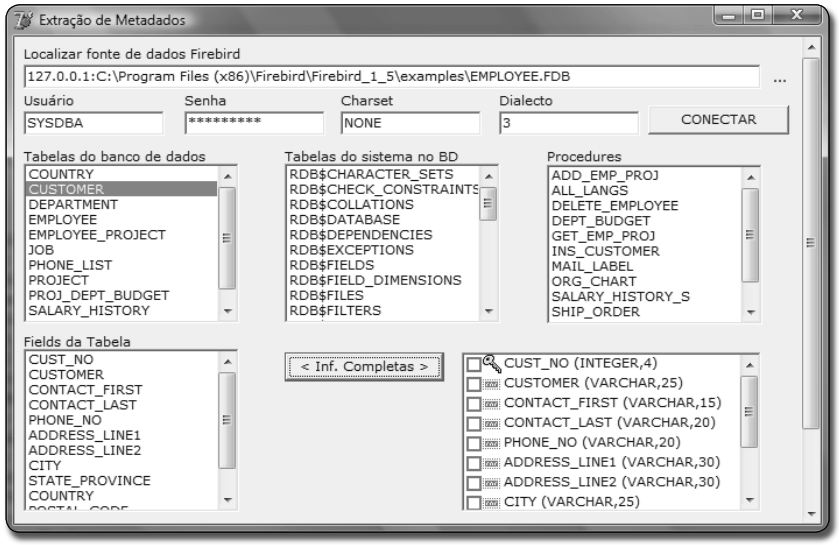


Figura 4. Informações de metadados da base selecionada

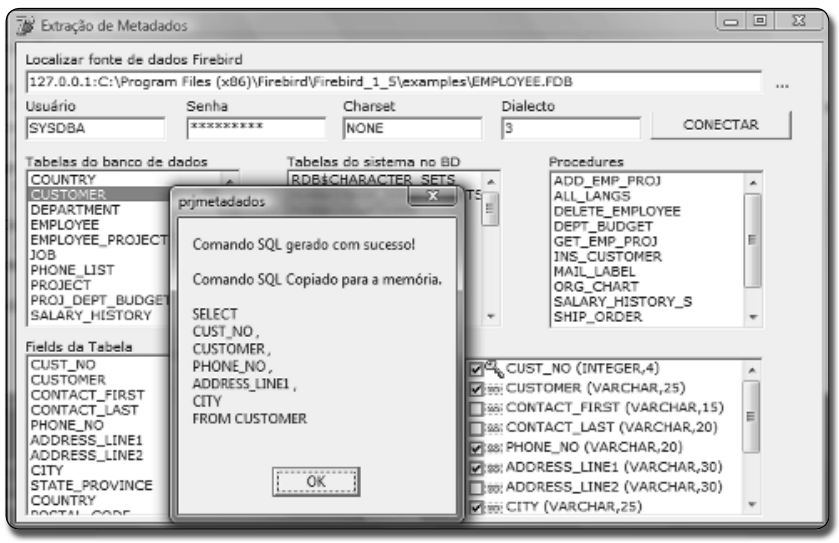


Figura 5. Montagem de instrução Select

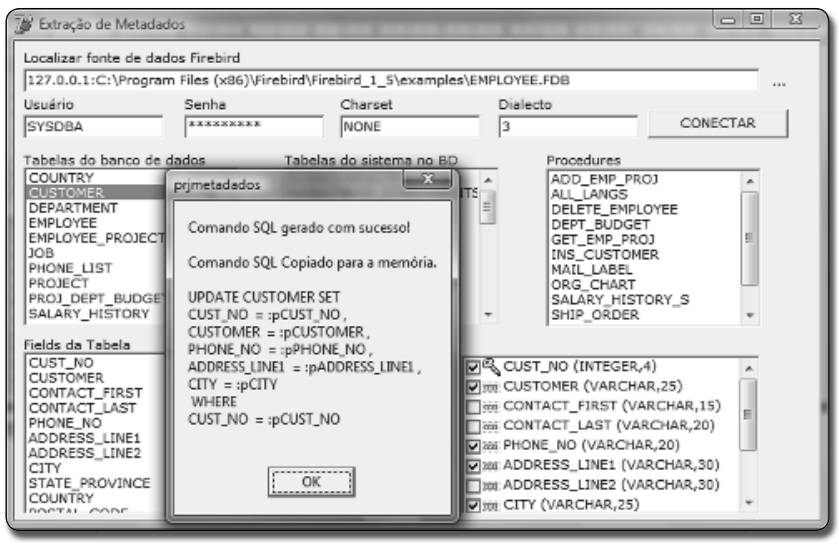


Figura 6. Montagem de instrução Update

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

Criando um componente de pesquisa

Reutilize as telas de pesquisa em sua aplicação

Resumo DevMan

O reaproveitamento de código é a preocupação de praticamente todo desenvolvedor. Evitar redigitar o mesmo código diversas vezes no mesmo sistema ou pior, replicar para outros produtos da empresa é uma perda de tempo para toda e qualquer equipe. Portanto, o melhor a se fazer é tentar sanar esse problema utilizando-se de artifícios bem comuns de toda linguagem de desenvolvimento.

A criação de funções, dlls e componentes é sem dúvida a melhor opção. Nesse artigo, veremos como criar um componente de pesquisa para ser utilizado em todo o sistema. Dessa forma, evitamos redigitação desnecessária de código em partes distintas do sistema.

Nesse artigo veremos

• Criação de componentes;

• Criação de pacote para conter nossos componentes.

Qual a finalidade

• Criar um componente de pesquisa totalmente funcional que poderá ser utilizado em qualquer parte do sistema.

Quais situações utilizam esses recursos?

• Em qualquer aplicação que utiliza pesquisas, praticamente em toda aplicação pode ser usado esse tipo de recurso.

Luciano Pimenta

lucianoalmeidapimenta@gmail.com

é Técnico em Processamento de Dados, desenvolvedor Delphi/C#. Palestrante da 4ª edição da Borland Conference (BorCon). Autor de mais de 50 artigos e de mais de 250 vídeo-aulas publicadas em revistas e sites especializados. Atualmente é programador Web da FullSoft - Mobile Solutions em Caxias do Sul-RS. Blog: lucianopimenta.blogspot.com. Site: www.lucianopimenta.net. É consultor da FP2 Tecnologia (www.fp2.com.br).

Em uma aplicação, a quantidade de consultas auxiliares é bastante grande, ou seja, pesquisa de clientes, fornecedores, produtos, etc., além de serem utilizadas em vários locais da aplicação. Criar um formulário de consulta para cada uma das pesquisas é trabalhoso, então por que não criar um componente para essas consultas? Certamente, posso citar uma infinidade de motivos para se criar um componente para efetuar pesquisas, alguns dos principais motivos são:

• *Encapsulamento:* encapsulamento é uma das práticas, em minha opinião, mais interessantes em programação OO, já que o código fica todo isolado em uma classe, componente, biblioteca etc., fazendo com que tenhamos mais controle sobre o que ocorre na aplicação;

• *Reaproveitamento de Código:* utilizar o mesmo código em diversos lugares da aplica-

ção com certeza é dar um *tiro no pé*. Estando o código em um único repositório, temos a certeza de que após uma possível manutenção nele, toda a aplicação sofrerá a alteração;

O que vou mostrar neste artigo, é um componente simples, para localizar dados, que pode ser modificado de acordo com a necessidade do desenvolvedor.

Iniciando a construção

A primeira providência a tomarmos é criar um pacote para alocar os arquivos de nosso componente. É importante criarmos um pacote, pois a partir da versão 2006 do Delphi, não há como instalarmos componentes utilizando apenas o *.pas* do mesmo. Além disso, nosso componente conterá um formulário para efetuar as pesquisas e devolver o resultado. Nesse artigo utilizaremos o Delphi 7, porém essa mesma prática pode ser realizada em qualquer outra versão de sua escolha.

Para criar um pacote clique em *File>New>Other>New>Package* conforme podemos visualizar na **Figura 1**. Após a criação do pacote salve o mesmo com o nome *pkLocalizar.dpk* utilizando o menu *File>Save*.

Criando o formulário

Teremos de criar o formulário de pesquisa, que deve conter os componentes para digitar o texto, botão de pesquisa e um *grid* onde serão mostrados os dados, além de um *DataSource* que será vinculado a um *DataSet*. Você pode incluir componentes de sua necessidade. A idéia da criação do formulário é bem simples. Ele se tornará padrão em nosso sistema, ou seja, toda tela que necessitar de uma pesquisa fará uso do componente e conseqüentemente da janela padrão de pesquisa, dando um ar profissional à aplicação.

Entre no menu *File>New>Form* e salve-o como *frmPesquisa.pas* e em sua propriedade *Name* digite *frLocalizar*. Insira alguns componentes de maneira que o layout da tela se pareça com a **Figura 2**.

No código do formulário devemos criar alguns métodos e propriedades, conforme o código a seguir:

```
procedure Buscar;
function GetCodigo: integer;
property DataSet: TClientDataSet;
```

No código anterior, criamos um método que será responsável por realizar a consulta (*Buscar*) e uma função. Além disso, temos que criar uma propriedade do tipo *TClientDataSet*, responsável por ter os dados da consulta.

Caso deseje, mude para o controle que utiliza para trabalhar com dados locais (*IBQuery*, *Query* etc.). O código do *Buscar*, simplesmente, repassa para o *ClientDataSet* o valor do *Edit1*, onde o usuário digitará o valor de busca:

```
FDataSet.Close;
FDataSet.Params[0].AsString := '%' + Edit1.
Text + '%';
FDataSet.Open;
```

O código para retornar o campo chave da tabela é bastante simples, onde retornamos o primeiro campo (*Field*) do *ClientDataSet*:

```
Result := DataSet.Fields[0].AsInteger;
```

Claro, precisamos indicar o campo chave na consulta SQL do *ClientDataSet* para que seja o primeiro registro. Ao apertar *Crtl+ Shift + C*, o Delphi cria o manipulador para a propriedade *DataSet*. O código do *SetDataSet* fica conforme o código a seguir:

```
if FDataSet <> Value then
begin
  FDataSet := Value;
  DataSource1.DataSet := FDataSet;
end;
```

No código, atribuímos a propriedade *DataSet* do *DataSource*, o valor da propriedade. Assim, ao configurarmos a propriedade no *Object Inspector*, o *DataSet* é vinculado ao *DataSource*.

Codificando o formulário

Para os botões, que vão indicar se devemos confirmar o item selecionado no *DBGrid* ou cancelar o formulário (fechando-o), devemos chamar o *Close*, e a seguir, configurar o *ModalResult*, que para o botão OK, será *mrOK* e *Cancelar* como *mrCancel*.

Veremos adiante, o motivo de usarmos o *ModalResult*. No evento *OnDblClick* do *DBGrid*, devemos usar o mesmo código do botão OK, apenas verificando antes se existem registros na consulta, verificando a propriedade *RecordCount* do *FDataSet*. Assim, o usuário pode dar um duplo clique no *grid* e o mesmo será fechado.

No *OnDataChange* do *DataSource*, vamos habilitar o OK, apenas se existirem registros no *ClientDataSet*, ou seja, mais uma validação que fizemos para não termos problemas com o componente:

```
btnOk.Enabled := FDataSet.RecordCount > 0;
```

Nota do DevMan

Até a versão 7.0 do Delphi era possível fazer a instalação de componentes apenas utilizando o arquivo .pas ou .dcu do componente. Já no Delphi 8.0 em diante esse recurso (*Component>Install component*) não está mais acessível. Agora, é necessário criar um novo pacote utilizando o menu File>New>Package – Delphi for Win32. Em seguida salve o pacote e adicione o arquivo .pas ou .dcu do componente que deseja instalar. Salve novamente o pacote, compile e instale.

A partir de agora, sempre que quiser instalar esse mesmo componente, basta abrir o pacote, compilar e instalar.

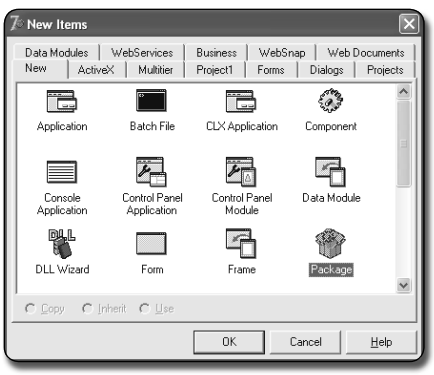


Figura 1. Criando o pacote do componente

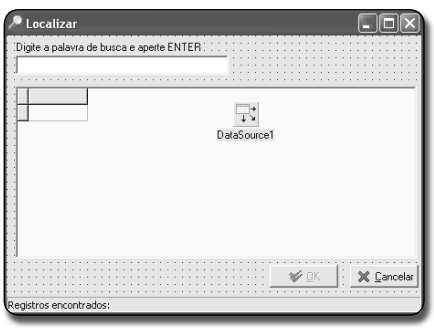


Figura 2. Formulário de pesquisa

Para finalizar o formulário, precisamos tratar o evento *OnKeyPress* do *Edit*, usando o código da **Listagem 1**.

No código anterior, primeiramente, verificamos se foi apertada a tecla *Enter*. Após, verificamos se o *Edit* não está vazio, isso é bom para que uma pesquisa muito grande não seja executada, trazendo lentidão ao sistema.

Poderíamos ainda, colocar uma quantidade mínima de caracteres a serem digitados para que o usuário chegue o mais perto possível do que está procurando. Após verificarmos se a propriedade *DataSet* não está vazia, chamamos o método *Buscar*, configuramos o foco para o *DbGrid* e adicionamos uma mensagem na *StatusBar*, com a quantidade de registros encontramos com o valor digitado no *Edit*.

O *key := #0* não emite o bip característico do sistema operacional. Com essa configuração, o formulário de consulta está pronto. Alguns retoques a mais poderiam ser feitos, como configurar as colunas do *Grid* para que fiquem centralizadas pelo código e alinhadas à esquerda com o texto.

Listagem 1. Código do OnKeyPress do Edit

```
{ se apertou ENTER, realiza a busca }
if key = #13 then
  if Edit1.Text <> '' then
  begin
    if Assigned(FDataSet)then
    begin
      Buscar;
      DBGrid1.SetFocus;
      StatusBar1.SimpleText := Format(
        '%d registro(s) encontrado(s) com "%s"',
        [FDataSet.RecordCount, Edit1.Text]);
      Key := #0;
    end;
  end;
```

Listagem 2. Unit do componente

```
unit uLocalizar;

interface

uses Classes, Forms, DB, Dialogs, ufrLocalizar, Graphics, Controls,DBClient;

  type
   TLocalizar = class(TComponent)
   private
    FDataSet: TClientDataSet;
    FFormLocalizar: TfrLocalizar;
    procedure SetDataSet(const Value: TClientDataSet);
   protected
    procedure Notification(AComponent: TComponent;
      Operation: TOperation); override;
   public
     function Execute: Boolean;
     function ReturnValue: integer;
   published
    property DataSet: TClientDataSet read FDataSet write SetDataSet;
  end;

procedure Register;

implementation

procedure Register;
begin
  RegisterComponents('Localizar Component',
    [TLocalizar]);
end;
```

Listagem 3. Método Execute

```
if not FDataSet.Active then
  FDataSet.Active := True;
FFormLocalizar := TfrLocalizar.Create(nil);
try
  FFormLocalizar.DataSet := FDataSet;
  FFormLocalizar.ShowModal;
  Result := FFormLocalizar.ModalResult = mrOk;
finally
  FFormLocalizar.Free;
end;
```

Quando o usuário apertar ESC, poderíamos fechar o formulário, ou ainda, quando o usuário apertar *Enter* no *grid* o mesmo fechar como se fosse clicado o botão OK. Isso são funcionalidades simples, que deixarei a cargo do leitor, mas a versão para download possui essas funcionalidades.

Criando o componente

Agora, precisamos criar uma *unit*, que receberá mais propriedades que serão repassadas ao formulário no momento de sua criação em memória, assim como também fazer o registro do componente no Delphi. Crie uma nova *unit* e adicione o código da **Listagem 2**.

No código, criamos uma classe *TLocalizar*, descendente de *TComponent*. Na seção *private* temos duas variáveis, uma referente à propriedade publicada *DataSet* (seção *published*) que teremos e outra referente ao formulário criado neste artigo.

O *SetDataSet* configura a propriedade e sua implementação é bem simples, onde apenas precisamos configurar a propriedade *DataSet* e verificar se o formulário está criado, através do *FFormLocalizar*.

Na seção *protected*, precisamos implementar o *Notification*, que é responsável, por atribuir *nil* à propriedade *DataSet*, quando por exemplo o *ClientDataSet* associado a mesma, for excluído. Isso evita erros no IDE do Delphi. A sua implementação, usa o seguinte código:

```
inherited;
if Operation = opRemove then
  if AComponent = FDataSet then
    FDataSet := nil;
```

Continuando o código da **Listagem 2**, na seção *public* temos duas funções. A primeira, *Execute*, indicará se houve algum registro escolhido no formulário, assim podemos repassar para a consulta principal, o valor do registro. A segunda retorna o valor da chave primária do registro escolhido no *grid* do formulário, onde simplesmente, chama o *GetCodigo* da variável *FFormLocalizar*. Por fim, o *Register* (abaixo de *implemetation*), registra o componente na *Component Pallete* do Delphi, onde

passamos como parâmetro o nome a paleta e a classe do componente.

Chamando o formulário de pesquisa

O *Execute*, é o principal método do nosso componente. Ele se comporta semelhante ao *Execute* dos controles *Dialogs* do Delphi, ou seja, é retornado *true* se foi escolhido um registro na consulta. Na **Listagem 3**, temos a implementação do método.

O código é bastante simples, primeiro verificamos se o *ClientDataSet* vinculado ao componente está ativo, se não abrimos o mesmo. A seguir, instanciamos o formulário de pesquisa, repassamos a propriedade *DataSet* e chamamos o *ShowModal*.

O resultado do *Execute*, será de acordo com o *ModalResult* do formulário, ou seja, caso seja *mrOK* foi selecionado algum registro na consulta, senão, nenhum registro foi selecionado. Por isso, configuramos nos botões do formulário o *ModalResult*.

Por fim, no *finally*, liberamos o formulário da memória. Para finalizar, toda a criação do componente, no pacote, clique em *Compile* e depois em *Install*. Caso tudo tenha dado certo, você terá um componente instalado em uma nova paleta no Delphi (**Figura 3**).

Colocando um ícone no componente

Para adicionarmos um ícone ao componente, precisamos editar um arquivo DCR e adicionar no pacote. Vamos usar o *Image Editor* do Delphi, acessando o menu *Tools>Image Editor*. No editor, clique em *File>New>Component resource file (dcr)*.

Clique com o botão direito em *Contents* e escolha *New>Bitmap*. Escolha uma imagem de tamanho 24x24 (use *SuperVGA* nas opções de cores). Após criar, dê um duplo clique *Bitmap1* e crie ou cole uma imagem de sua preferência. Após adicionar a imagem, renomeie o arquivo *Bitmap1* para *TLOCALIZAR*. Feche o editor, salvando o arquivo na mesma pasta do pacote criado no Delphi. Agora no *source* do pacote (DPK), adicione o código que irá adicionar o arquivo:

```
{$R *.dcr}
```

Agora, basta recompilar a aplicação e seu componente terá um novo ícone (**Figura 4**).

Testando o componente

Agora, precisamos testar o componente. Crie uma conexão com um banco de dados, usando *dbExpress* ou qualquer outra suíte de componentes para acesso a dados. Nosso principal componente é o *ClientDataSet*.

Adicione um *SQLDataSet* fazendo a ligação ao *SQLConnection* que conecta ao banco. Adicione também um *DataSetProvider* e modifique para *True* a propriedade *Options>AllowCommandText*, assim indicamos que o responsável por receber as consultas (instruções SQL) é o próprio *ClientDataSet*.

Adicione dois *ClientDataSets* ao formulário (*cdsEmployee* e *cdsBuscaEmployee*). No *cdsEmployee* configure o *CommandText* para:

```
select * from EMPLOYEE where EMP_NO = :EMP_NO
```

Estamos selecionando todos os dados da tabela, mas parametrizando a mesma pelo código, assim nossa consulta retornará todos os registros, apenas quando indicarmos o valor do parâmetro.

No *cdsBuscaEmployee* digite na mesma propriedade:

```
select EMP_NO, FIRST_NAME from EMPLOYEE
where FIRST_NAME like :FIRST_NAME
```

Nesse controle, estamos retornando apenas o código e nome da tabela *Employee*, parametrizando pelo nome, que será configurado no nosso componente. Não esqueça de configurar os parâmetros nos dois *ClientDataSets*. Adicione um *Localizar* no formulário e configure *DataSet* para *cdsBuscaEmployee*. Adicione os campos do *cdsEmployee* no formulário para que o mesmo fique semelhante à

ClubeDelphi PLUS www.devmedia.com.br/clubedelphi/portal.asp

Acesse agora o mesmo o portal do assinante ClubeDelphi e assista a uma vídeo aula de Guinther Pauli que mostra como criar componentes.

*www.devmedia.com.br/articles/viewcomp.asp?comp=7506&hl=*cria%E7%E3o*%20and%20*de*%20and%20*componentes**

Figura 3. Componente instalado no Delphi

Figura 4. Novo ícone do componente

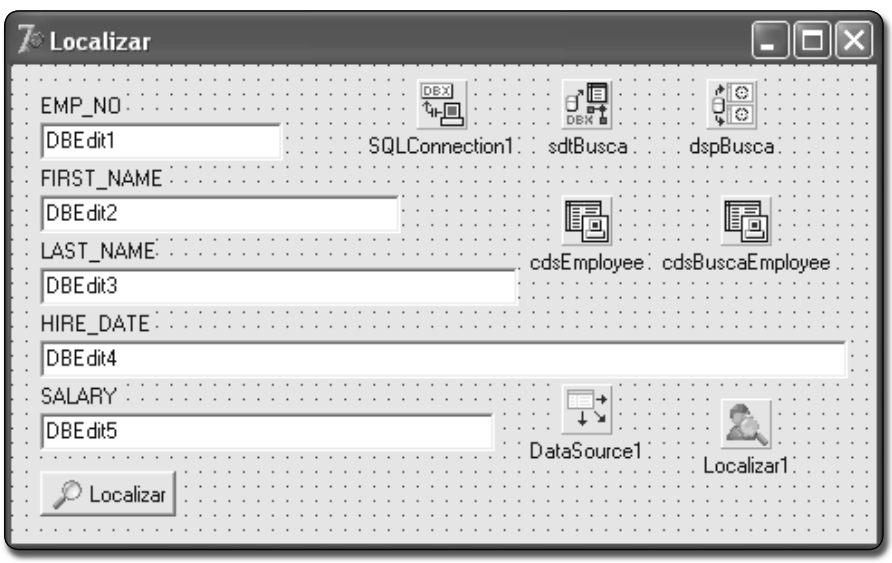


Figura 5. Formulário de cadastro para teste

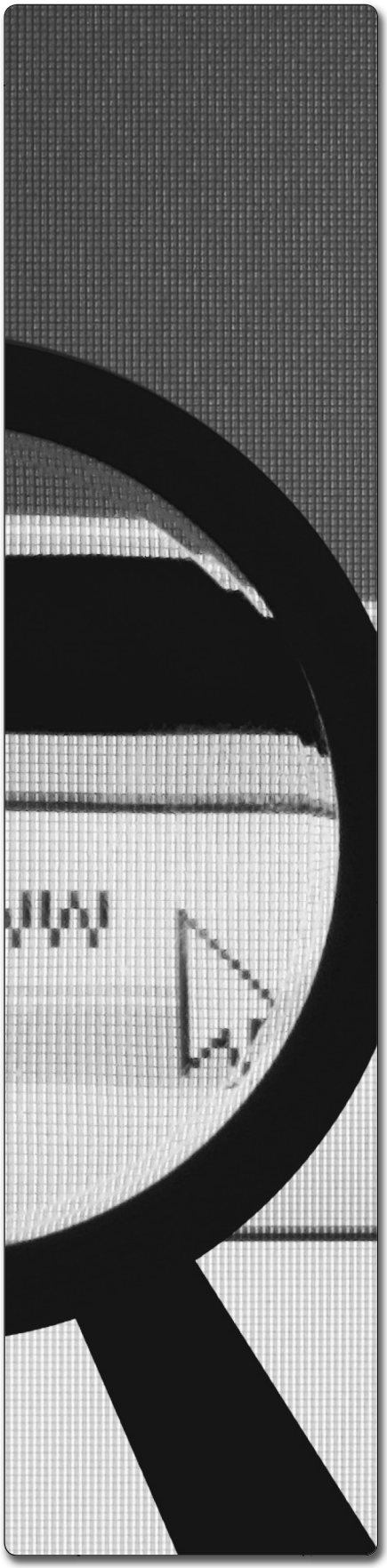


Figura 5. Adicione também um botão para chamar o *Localizar*.

Agora, simplesmente, precisamos usar o seguinte código no botão:

```
if Localizar1.Execute then
begin
  cdsEmployee.Close;
  cdsEmployee.Params[0].AsInteger :=
    Localizar1.ReturnValue;
  cdsEmployee.Open;
end;
```

Veja que o código é bem simples, onde verificamos se foi escolhido algum registro no formulário de consulta, usando o *Execute*. Após, simplesmente, configuramos o parâmetro da consulta com o valor retornado pelo componente (*ReturnValue*).

Com isso, temos um componente de pesquisa que pode ser reaproveitado em vários formulários usando o mesmo componente de dados. Veja o mesmo em execução na **Figura 6**.

Sugestões de melhorias

Imagine criar uma propriedade que seria uma coleção de parâmetros com *Caption*, campo da consulta etc. Ao adicionar um item na coleção, o mesmo seria vinculado a um item de um *RadioButton* no formulário, assim, para cada parâmetro teríamos um item em um componente para escolher e o campo onde faríamos a busca.

Teríamos assim, um componente de busca bastante customizável, criando consultas complexas com vários parâmetros. Também poderíamos melhorar o visual do formulário de consulta, trazendo facilidades ao usuário.

Caso implemente uma funcionalidade que queira divulgar para outros programadores, mande suas alterações que publicarei em meu site (*www.lucianopimenta.net*) e no da ClubeDelphi.

Conclusão

Vimos neste artigo como podemos criar componentes que nos auxiliem em aplicações client/Server para ser utilizado em consultas auxiliares, poupando tempo de programação e diminuição de código.

O que mostrei aqui foi apenas o pontapé inicial, use sua criatividade para agilizar seu dia a dia de programador. Um grande abraço a todos! ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

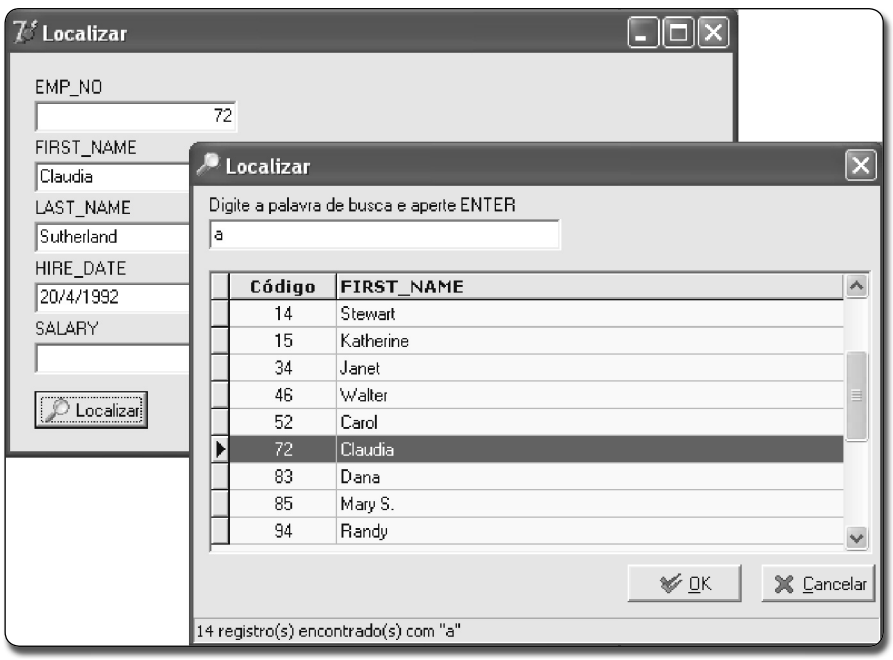


Figura 6. Testando o componente

Já pensou em hospedar todos os seus sites em um único plano de hospedagem?

PLANO PROFISSIONAL 1

30 DIAS GRÁTIS

Ao assinar, digite o código de desconto exclusivo para leitores desta revista e ganhe!

RVSTMEDIA

- 3 domínios independentes
- 50 GB de transferência
- 30 GB de e-mails com SSL
- 1 GB de espaço
- Painel de controle
- ASP, ASP.NET 3.5 E PHP 5
- Access ilimitado
- 3 bancos MYSQL 5
- 1 banco SQL Server 2005 Express

Tudo isso por apenas R$ **25,90** por mês

Na Hospedix você ainda ganha registro de domínio (.com .net .org ou .info) grátis e isento de taxas anuais enquanto for cliente.

Acesse agora mesmo e descubra por que Hospedix é a hospedagem de sites preferida entre os profissionais.

hospedix.com.br

HOSPEDIX
HOSPEDAGEM PROFISSIONAL

Nesta seção você encontra artigos intermediários sobre Delphi Win32 e Delphi for .NET

Controle de Hardware com Delphi

Usando um LCD no Delphi

Resumo DevMan

Durante a época do MS-DOS e Windows 3.1, 95 e 98 tínhamos como desenvolver facilmente programas para envio de informações ao hardware utilizando diversas linguagens. Uma delas era o próprio Delphi em sua primeira versão. A utilização da porta paralela para interfacear hardware era comum, porém, hoje, nos deparamos com um grande problema: a dificuldade em se enviar comandos para a porta paralela usando programas de modo usuário.

Nesse artigo veremos como fazer uso do Delphi e do componente LPTPort para criar um exemplo que manipula um display de LCD.

Nesse artigo veremos

- Breve histórico de acesso ao hardware em diferentes versões do Windows;
- Acesso ao hardware através de funções em Assembly;
- Acesso ao hardware através de dlls como o InpOut32.dll;
- Detalhes sobre LCDs (Arquitetura e Funcionamento);
- Construção de uma classe de controle de LCDs (THD44780);
- Projeto de um programa de Controle de LCDs.

Qual a finalidade

- Controle e manipulação de LCDs do tipo HD44780 pela porta paralela.

Quais situações utilizam esses recursos?

- CaseModding, hobbistas de eletrônica e projetos de interfaceamento de dispositivos eletrônicos.

Thiago Batista Limeira

tblimeira@gmail.com

É engenheiro da computação e trabalha com Delphi há mais de 4 anos, atualmente desenvolve programas e interfaces homem-máquina que envolvam comunicação de dados. Pode ser contatado pelo endereço tblimeira@gmail.com.

Engenheiros, *hobbistas* e outros interessados em utilizar o computador para manipular *hardware* externo, durante muito tempo utilizaram o *Turbo Pascal* e o *Delphi 1*, pois estas ferramentas possuíam uma combinação de poder e velocidade de compilação, facilitando o interfaceamento de circuitos e outros dispositivos eletrônicos. Esta flexibilidade e poder de programação eram ótimos em *DOS*, também no *Windows 3.1 e Windows 95* com o *Delphi 1*, que possuía comandos de acesso as portas do PC. Mas a partir do *Delphi 2* em diante, juntamente com o *Windows NT*, a manipulação direta do *hardware* não é mais permitida e devido a isto o acesso nativo ao *hardware* pelas linguagens

visuais modernas foi retirada.

Neste artigo faremos uma revisão dos métodos mais utilizados para acesso ao *hardware* através do *Delphi*, e mostrar todas as formas utilizadas de acesso ao *hardware* através da evolução do *Windows* e do *Delphi*. E para demonstrar as capacidades de controle do *Delphi* controlaremos um *LCD* de *16x2* pela porta paralela (**Figura 1**) utilizando o componente *LPTPort*.

A saga do acesso ao hardware

No final dos anos 80, caso alguém desejasse acender *LEDs* através do computador teria de ligá-los a um cabo de impressora, e escrever um programa que alterasse o valor armazenado em um endereço específico de memória (no caso o endereço da porta paralela). Em *Turbo Pascal* era possível utilizar o comando *Port* a seguir:

```
Port[Endereço] := Valor;
```

Este comando funcionava sem dificuldades nas primeiras versões do *Windows e Delphi 1*, já que o *Delphi 1* era *16 bits* e até meados de 1995 ainda possuía um comando *Port* de *8 bits* e *PortW* de *16 bits*. Então o *Windows* ficou "melhor". Acesso direto a localizações específicas se tornou uma má idéia, e a partir do *Delphi 2* o comando foi retirado. Apesar de não ser mais comercializado, o *Delphi 1* acompanha versões subseqüentes do *Delphi*. Em sua primeira versão é possível produzir programas que rodam nas versões *3.1, 95* e *98* do *Windows*, porém nada de *Windows NT 4.0*.

A partir da versão 2 em diante, ainda era possível acessar o *hardware* do computador de forma direta através de instruções em *assembly* como na **Listagem 1**.

> **Nota**: Essas funções, *WritePort* e *ReadPort*, funcionavam apenas nas versões do Windows 95 e 98.

A título de curiosidade, veja que o cabeçalho das funções é Delphi, mas o corpo em si é todo *Assembly*. Inclusive diversas funções que utilizamos no dia-a-dia são desenvolvidas em *Assembly*, como *IntToStr*, por exemplo.

A era das restrições

Como vimos, escrever programas para controlar a porta paralela no tempo do velho *MS-DOS* era muito simples, assim como na época do *Windows 95/98*. Podíamos utilizar as funções *WritePort* e *ReadPort* sem nenhuma restrição. Mas ao entrar na nova era do *Windows NT, 2000* e *XP*, toda esta simplicidade foi retirada. E com certeza, se em alguma ocasião você já tentou controlar as portas do PC em *Windows NT*, deve ter se deparado com problemas.

Quando tentamos rodar um programa que utiliza funções convencionais de acesso ao *hardware* nestas versões *Windows*, o sistema operacional mostrará uma mensagem de exceção de instrução privilegiada (**Figura 2**).

Sendo um sistema operacional seguro, o *Windows NT* confere alguns privilégios e restrições aos diferentes tipos de programa que operam sobre ele. Ele classifica todos os programas em duas categorias: modo *usuário* e modo *kernel* (**Figura 3**).

Os programas que geralmente criamos estão na categoria do modo usuário. Programas de modo usuário são restritos a certas instruções como entrada, saída, etc. Toda vez que um programa de modo usuário tenta executar tais instruções, o sistema operacional interrompe a execução deste programa e mostra uma mensagem de erro.

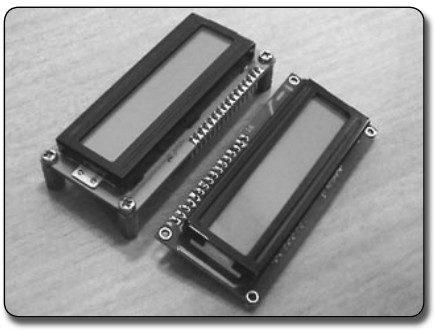


Figura 1. Interfaceamento de módulos LCD

Figura 2. Mensagem erro durante uma tentativa de acesso ao hardware

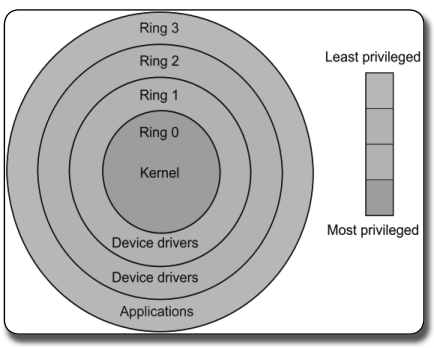


Figura 3. Arquitetura de acesso ao hardware em Windows NT/2000/XP

Nota do DevMan

A porta paralela é uma ferramenta de controle simples e barata para construção de dispositivos e projetos. A simplicidade e facilidade de programação fazem da comunicação paralela a mais popular dos *hobbistas* eletrônicos, ela também é utilizada em robôs controlados por computador, programadores de micro controladores, automação doméstica, etc, por isso sua utilização nesse artigo é indispensável, visto que conseguiremos elaborar um exemplo simples, mas bastante detalhado.

Listagem 1. Código para escrita e leitura na porta paralela nas versões do Windows 95 e 98

```
procedure WritePort(Port: word; Value: byte);
asm
 push word ptr Port
 mov al, Value
 pop dx
 out dx, al
end;

function ReadPort(Port: word): byte;
asm
 mov dx, Port
 in al, dx
end;
```

Em resumo: nossos programas de interfaceamento são interrompidos quando executam instruções de entrada ou saída para leitura ou escrita de dados na porta paralela. Mas ao mesmo tempo programas de modo *kernel* não são restritos de nenhuma forma ao executar tais instruções. *Device drivers* são capazes de rodar em modo *kernel*. Então para solução do problema de acesso ao *hardware* se resume em escrever um *driver* de modo *kernel* capaz de ler e escrever dados na porta e permitir ao programa de modo usuário comunicar-se com ele.

Só que escrever um *driver* não é tarefa nada fácil, mesmo para os programadores mais experientes. Então para obter acesso ao hardware necessitamos utilizar *drivers* (*DLLs*). Sinceramente uma das vantagens, ou pontos fortes do *Delphi* é que, normalmente, não usamos *DLLs*, mas realmente não existe outra forma de acesso ao *hardware* que não seja feita através delas.

Mesmo os componentes de acesso ao *hardware* como o *LPTPort* fazem uso de *drivers* para obter acesso direto ao equipamento.

Acessando o hardware com o uso de DLL

Uma das *DLLs* mais utilizadas para acesso direto ao *hardware* é a *InpOut32.dll*, que pode facilmente ser baixada no site do desenvolvedor em *www.logix4u.net*. A **Listagem 2** apresenta um exemplo de utilização da *InpOut32.dll* em uma *unit* no *Delphi*.

A *InpOut32.dll* possui as seguintes características:

- Funciona nas versões *do Windows 98/NT/2000* e *XP*;
- Usa um *driver kernel* encapsulada em uma *DLL*;
- Não requer nenhuma instalação de *software* ou *driver*;
- O *driver* é automaticamente instalado e configurado quando a *DLL* é carregada;
- Nenhuma *API* especial é necessária, pois são apenas duas funções principais (*Inp32* e *Out32*);
- Pode ser facilmente utilizada com *Delphi*.

Para importar métodos ou funções de *DLLs* devemos utilizar a palavra reservada *external* para informar que se trata de uma função externa e a diretiva *stdcall* é um comando que permite que outras linguagens possam fazer chamadas às funções.

Agora que fizemos uma breve recapitulação sobre formas de acesso ao *hardware* iremos programar nosso projeto de controle de *hardware*.

Detalhes importantes sobre LCDs

LCDs são muito utilizados em interfaces eletrônicas, para indicar o estado de diversos equipamentos. Além disso, eles têm sido extensivamente utilizados em *CaseMod*.

Existem diversos projetos de como se utilizar um *LCD* para mostrar diversas informações do PC como velocidade e temperatura do processador, o horário local, a lista de músicas do *Winamp*, etc. Inclusive existem programas especializados em controle de *LCD* como o *LCDSmartie* que é *opensource* e que foi totalmente desenvolvido em *Delphi*.

Listagem 2. Código exemplo de utilização da InpOut32.dll no Delphi

```
unit InpOut32;

interface

uses
SysUtils, Windows, Messages, Classes;

implementation

{ Definição das funções da InpOut32.dll }
function InPortb(EndPorta: integer): byte stdcall; external 'inpout32.DLL' name 'Inp32';

procedure OutPortb(EndPorta: integer; Valor:byte); stdcall;
  external 'inpout32.DLL' name 'Out32';
end.
```

Pino	Nome	Função
1	Vss	Terra
2	Vdd	5V
3	Vee	Tensão de Contraste
4	RS	Seleção de Registro (Register Select)
5	R/W	Leitura/Escrita (Read/Write)
6	E	Habilitar (Enable/Clock)
7	D0	Bit de dados 1
8	D1	Bit de dados 2
9	D2	Bit de dados 3
10	D3	Bit de dados 4
11	D4	Bit de dados 5
12	D5	Bit de dados 6
13	D6	Bit de dados 7
14	D7	Bit de dados 8
15	A	Anodo do BackLight (se houver)
16	K	Catodo do BackLight (se houver)

Tabela 1. Tabela Pinagem LCD HD44780 ou compatível

Porta Paralela	LCD HD44780
D0 (Pino 2)	D0 (Pino 7)
D1 (Pino 3)	D1 (Pino 8)
D2 (Pino 4)	D2 (Pino 9)
D3 (Pino 5)	D3 (Pino 10)
D4 (Pino 6)	D4 (Pino 11)
D5 (Pino 7)	D5 (Pino 12)
D6 (Pino 8)	D6 (Pino 13)
D7 (Pino 9)	D7 (Pino 14)
Strobe (Pino 1)	Enable (Pino 6)
Error (Pino 14)	Read/Write (Pino 5)
Select In (Pino 16)	Register Select (Pino 4)

Tabela 2. Tabela de ligação Porta Paralela - LCD HD44780

Desenvolveremos um programa para controlar um *LCD 16x2* com o qual poderemos realizar a inicialização do *display*, escrever textos, enviar comandos, etc. Tudo isto através da porta paralela do PC e utilizando o componente *LPTPort* que na minha opinião oferece o método mais simples e fácil de acesso à porta paralela.

Antes de iniciarmos a programação do *LCD*, devemos antes conhecer alguns detalhes importantes que serão necessários para a execução deste projeto, entre eles qual tipo de *LCD* utilizaremos, qual a *pinagem* do *display*, esquema de ligação, comandos, etc.

Utilizaremos um módulo *LCD* alfanumérico com um controlador *Hitachi HD44780* (ou compatível), pois este tipo de módulo pode ser mais facilmente encontrado. O controlador do *display* é encarregado de controlar a matriz de cristal líquido. É óbvio que este tipo de módulo não oferece as mesmas vantagens de módulos mais modernos que são utilizados em celulares, *laptops* e muitos outros equipamentos, mas sua simplicidade será uma vantagem no momento da montagem do mesmo. Os módulos de *LCD* possuem um padrão de pinos. Normalmente os que não têm *backlight* (luz de fundo) possuem 14 pinos e os que possuem *backlight* têm 16 pinos. Esses dois pinos a mais são responsáveis pela ativação da luz de fundo. A **Tabela 1** mostra em detalhes os pinos de um módulo *LCD 44780* e suas respectivas funções.

Os pinos 1 e 2., *Terra* e *5V* respectivamente, são utilizados para energizar o módulo. O pino 3 é utilizado para controlar o contraste do *display*. O ideal é que este pino esteja conectado a uma fonte variável de tensão, para que dessa forma possamos variar a tensão e conseqüentemente o contraste. O pino 4 é o *Seletor de Registro* (*Register Select*) que é o primeiro de 3 pinos de controle.

Quando este pino está em baixa tensão (0V) então todos os *bytes* enviados ao *display* são tratados como comandos, e *bytes* lidos indicam seu estado. Ao colocarmos este pino em nível alto (5V) os dados de caracteres podem ser transmitidos do/para módulo.

O pino 5 é o *Leitura/Escrita* (*Read/Write*). Este pino deve ser colocado em nível baixo para escrever comandos ou caracteres no *display*, em nível alto para ler caracteres ou informação de status dos registros. O pino 6 é o *Habilitar* (*Enable*), utilizado para iniciar a transmissão de comandos ou caracteres dos pinos de dados. Quando escrevemos no *display*, os dados apenas são transmitidos na transição de nível alto para baixo deste pino. Porém quando lemos, os dados estarão disponíveis depois da transição de alto para nível baixo e outra transição de baixo para alto novamente.

E para finalizar, os pinos de 7 a 14 são os pinos de dados que podem ser transmitidos como um *byte* (8 bits) ou dois *nibbles* (4bits), veja o diagrama elétrico do display na **Figura 4**.

Dicas de montagem

Para aqueles que não possuem nenhuma ou pouca experiência em eletrônica, recomendo a leitura dos tutoriais do site *www.casemodbr.com* onde é possível encontrar uma lista de materiais e ferramentas necessárias para a construção de um cabo para interfacear o *LCD* a porta paralela. A **Tabela 2** mostra as ligações necessárias que o leitor deve fazer para ligar e comandar o *LCD* pelo *Delphi*.

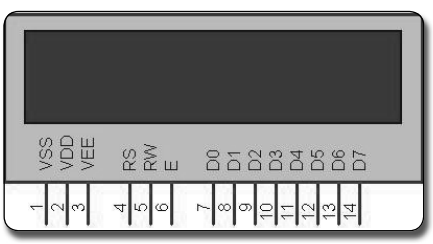


Figura 4. Diagrama elétrico de um LCD HD44780

> **Nota**: A montagem que utilizei neste artigo é uma montagem simplificada, visto que não iremos ler nenhum dado do *display*, sugiro que o leitor em sua montagem coloque o pino *Read/Write* (pino 5 do *LCD*) no terra. Desta forma não precisaremos nos preocupar em alterar o valor do pino 14 da porta paralela. Nosso exemplo se baseia em apenas escrever comandos e dados no *LCD* e só funcionará corretamente se o *Read/Write* estiver ligado ao terra.

Criando a classe THD44780

Agora que já entendemos o funcionamento básico do *display* iremos iniciar a construção de nossa aplicação de controle de *LCDs*. Para isso abra o *Delphi*, crie uma nova aplicação, que batizaremos

Nota do DevMan

A Porta Paralela está ligada diretamente à placa mãe de seu computador, por isso recomendo aos leitores que tenham muito cuidado ao conectar circuitos eletrônicos a essa porta, pois, uma descarga elétrica ou um componente com a polaridade invertida, poderá causar danos irreparáveis ao seu computador.

Todos nós da ClubeDelphi, não nos responsabilizamos por nenhum dano causado tanto no equipamento, como na montagem/instalação inadequada da interface.

Listagem 3. Definição da classe THD44780

```
type
  THD44780 = class
    private
    public
      Columns : integer;
      Lines   : integer;
      LPTPort : TLPTPort;
      procedure InitializeLCD;
      procedure ClearDisplay;
      procedure PosCursor(X, Y : integer);
      procedure Cursor_Display(CursorOn, DisplayOn, BlinkCursor : Boolean);
      procedure ShiftCursor_Message(ShiftText, ToRight : Boolean);
      procedure WriteText(Text : string);
      procedure SendCommand(Command : byte);
      procedure SendData(Data : byte);
      procedure WritePort(Address : word; Value : byte);
      constructor Create(LPT: TLPTPort);
      destructor  Destroy; override;
  end;
```

de *LCD.dpr*, e salve a *unit* como *uMain.pas*, coloque um componente *LPTPort* no *form* principal que será renomeado para *frmMain*. Criaremos inicialmente uma classe com a qual poderemos comandar o *LCD* mais facilmente. Crie uma nova *unit*, e salve-a com o nome de *uHD44780.pas*, nesta *unit* criaremos a classe de controle do *LCD* conforme a **Listagem 3**.

Com isso iremos encapsular todas as funcionalidades de controle de um *LCD* na classe *THD44780*, mais adiante, estaremos abordando cada um dos métodos desta nossa classe. Reparem que nossa classe possui algumas variáveis que podem ser visualizadas em seguida:

```
public
  Columns : integer;
  Lines   : integer;
  LPTPort : TLPTPort;
```

As variáveis *Columns* e *Lines* serão utilizadas para definir respectivamente o número de colunas e linhas de nosso *LCD*, nesse caso, em específico, estamos utilizando um *LCD 16x2* (16 colunas e 2 linhas). E para finalizar a variável *LPTPort*, que é uma variável do tipo *TLPTPort*, que irá realizar todo acesso à porta paralela.

Por enquanto declare a classe *THD44780* e tecle *Ctrl+Shift+C* para que o Delphi crie o cabeçalho das funções declaradas. A seguir, a **Listagem 4,** onde encontramos os métodos *Create* e *Destroy* da nossa classe *THD44780*.

> **Nota**: O componente *LTPPort* encontra-se disponível juntamente com o código-fonte desse artigo no site da ClubeDelphi.

Como podemos observar, esta classe quando criada, recebe o parâmetro *LPT*, que na verdade é apenas um ponteiro para uma variável do tipo *TLPTPort* que será o componente utilizado para acessar a porta paralela como já dito anteriormente. Depois configuramos o número de Colunas e Linhas através das variáveis *Columns* e *Lines*.

Quanto ao método *WritePort*, que pode ser visto nas duas últimas linhas do método *Create*, é apenas uma subrotina para configurar o endereço e o valor da porta, desta forma evitaremos utilizar 2 linhas de comando repetitivamente, o método *WritePort* pode ser visto a seguir:

```
procedure THD44780.WritePort(Address : word;
Value : byte);
begin
  LPTPort.SelectPort(Address);
  LPTPort.Out(Value);
end;
```

Estamos, portanto utilizando o método *WritePort* para colocar o endereço de dados e controle da porta em zero. E no método *Destroy* liberamos o *LPTPort*, que está associado com o *HD44780*, da memória além de destruir nossa classe de controle.

Comando	Valor Hexadecimal	Tempo de execução mínimo
Limpar display	01	1,6 milissegundos
Cursor no início	02 ou 03	40 microssegundos
Modo de Funcionamento	04 a 07	40 microssegundos
Display On/Off e Cursor	08 a 0F	40 microssegundos
Desloca Cursor/Display	10 a 1F	40 microssegundos
Modo de Utilização	20 a 3F	40 microssegundos
Configurar DDRAM	80 a FF	40 microssegundos

Tabela 3. Tabela de comandos do LCD HD44780

Listagem 4. Métodos Create e Destroy da classe THD44780

```
constructor THD44780.Create(LPT: TLPTPort);
begin
  LPTPort := LPT;
  Columns := 16;
  Lines := 2;
  WritePort($378, 0);
  WritePort($37A, 0);
end;

destructor THD44780.Destroy;
begin
  LPTPort.Free;
  inherited Destroy;
end;
```

Listagem 5. Constantes que serão utilizadas na classe THD44780

```
const
{ Comandos }
LCD_LIMPA     = $01;  // Limpa o Display                          00000001
LCD_HOME      = $02;  // Coloca o cursor na 1. linha e 1. coluna  00000010
LCD_ONOFF     = $08;  // Controle do display:                     00001000
LCD_ONOFF_CP  = $01;  // Cursor piscante ou não                   00000001
LCD_ONOFF_C   = $02;  // Liga cursor ou não                       00000010
LCD_ONOFF_D   = $04;  // Liga Display ou não                      00000100
LCD_DESLOCA   = $10;  // Deslocamento do Cursor:                  00010000
LCD_DESLOCA_D = $04;  // Desloca pra direita ou esquerda          00000100
LCD_DESLOCA_M = $08;  // Deslocar Mensagem ou o cursor            00001000
LCD_FUNCAO    = $20;  // Modo de Utilização:                      00100000
LCD_FUNCAO_M  = $04;  // Usar Matriz 5x10 ou 5x7                  00000100
LCD_FUNCAO_L  = $08;  // 2 ou mais linhas ou 1 linha              00001000
LCD_FUNCAO_B  = $10;  // 8 bits ou 4 bits                         00010000
LCD_DDRAM     = $80;  // Estabelece endereço da DDRAM             10000000
```

Ligando o LCD

Energize o *LCD*, e repare que o mesmo deve estar apresentando uma série de quadrados na linha superior do *LCD* (**Figura 5**).

Neste estado inicial do *LCD*, mesmo que enviemos algum texto, ele não poderá ser visualizado. É necessário enviar algum comando ao display para que ele possa inicializar. Agora necessitamos criar uma rotina que servirá para enviar um comando para o *LCD*, a **Tabela 3** mostra uma série de comandos possíveis do *HD44780*.

Para auxiliar a utilização do *LCD*, também utilizaremos diversas constantes que facilitarão o envio dos comandos. A lista das constantes pode ser vista na **Listagem 5.**

Enviando Comandos ao LCD

Agora faremos algumas rotinas básicas para envio de comandos e dados. Conheceremos em detalhes como manipular o *LCD*. Para enviar um comando necessitaremos de seguir alguns passos:

1. Colocar o pino *Register Select* em baixa (para que o *display* interprete o sinal enviado como comando ou instrução), neste momento o pino *Enable* deverá estar em alta (pois apenas durante a transição de nível alto para baixo deste pino é que o *display* irá de fato processar o comando) e aguardar certo tempo até que o *LCD* possa processar novos comandos ou dados (uma vez que existe um tempo mínimo necessário para cada tipo de comando (**Tabela 4**));

2. Enviar o comando que desejamos e novo *delay* (espera);

3. E para finalizar o envio do comando colocamos o *Enable* em baixa e aguardamos outro *delay* cujo tempo é baseado no comando processado.

Agora que já sabemos como enviar um comando ao *display*, podemos implementar o método *SendCommand* que está na **Listagem 6**.

Nota: Um detalhe muito importante, é que para cada comando recebido pelo LCD existe um tempo mínimo que devemos aguardar para que o LCD possa processar novos comandos ou dados, normalmente este tempo de resposta varia desde microssegundos até alguns milissegundos. No nosso caso não necessitamos de tanta precisão e resolução de tempo, podemos utilizar o método Sleep do Delphi que interrompe a execução de um programa por um determinado tempo especificado em milissegundos.

Os pinos de controle do *LCD* estão sendo controlados pelo registro de controle da porta, por isso inicialmente escrevemos o valor 0 no endereço $37A da porta, desta forma teremos o pino *Enable* em nível lógico alto (5V) e o pino *Register Select* em nível lógico baixo (0V). Assim o *display* interpretará o sinal enviado como um comando. Devemos aguardar 1 milissegundo para enviar o parâmetro *Command* ao registro de dados que corresponde ao registro de dados da porta, definido pelo endereço $378 e aguardamos mais 1 milissegundo para finalizar o envio do comando, fazendo com que o *Enable* fique em nível baixo e concretize o envio do comando. Repare que dependendo do comando enviado (se é um limpar *display*, ou cursor no início) existe a necessidade de aguardar um tempo de 1 ou 2 milissegundos.

Listagem 6. Método de envio de comandos do THD44780

```
procedure THD44780.SendCommand(Command : Byte);
begin
  {Register Select em baixo e Enable em alto}
  {Obs: o pino 1 (Strobe) é invertido no hardware da porta}

  { Envia 00000000 para registro de controle }
  WritePort($37A,0);
  { 1 milisegundo }
  Sleep(1);
  { Envia comando para registro de dados }
  WritePort($378, Command);
  Sleep(1);
  {Register Select em baixo e Enable em baixo}
  { Envia 00000001 para registro de controle }
  WritePort($37A,1);
  if (Command = LCD_LIMPA) or (Command = LCD_HOME) then
    { 2 milissegundos }
    Sleep(2)
  else
    Sleep(1);
end;
```

Listagem 7. Método de envio de dados para o THD44780

```
procedure THD44780.SendData(Data : Byte);
begin
  {Register Select em alto e Enable em alto}
  WritePort($37A,4); // envia 00000100 para registro de controle
  sleep(1);
  WritePort($378,Data); //envia dados para registro de dados
  sleep(1);
  {Register Select em alto e Enable em baixo}
  WritePort($37A,5); // envia 00000101 para registro de controle
  sleep(1);
end;
```

Para enviar dados ao *LCD*, necessitamos utilizar o pino *Register Select* em baixa para que o sinal enviado seja interpretado como um dado a ser mostrado no *display*. Segue o procedimento que devemos utilizar:

1. Colocar o pino *Register Select* em alta (para envio de dados), neste momento o pino *Enable* deverá estar em alta e aguardar certo tempo até que o *LCD* possa processar novos comandos ou dados;

2. Enviar os dados e novo tempo de espera;

3. E para finalizar o envio dos dados colocamos o *Enable* em baixa e aguardamos outro tempo.

Agora que já conhecemos a forma de envio de dados, iremos construir nossa rotina *SendData* de acordo com a **Listagem 7**.

No método *SendData*, teremos basicamente o mesmo padrão do *SendCommand* sendo a principal diferença o fato de mantermos o *Register Select* em nível alto com o *WritePort* no endereço $37A (para que o sinal da porta seja interpretado

Nota do DevMan

O registro de controle da porta paralela é composto pelos pinos 1 (C0), 14 (C1), 16 (C2) e 17 (C3), com exceção do pino 16, todos os demais estão invertidos no hardware da porta paralela, em outras palavras, quando estiverem em nível lógico baixo, eles apresentam tensão de 5V.

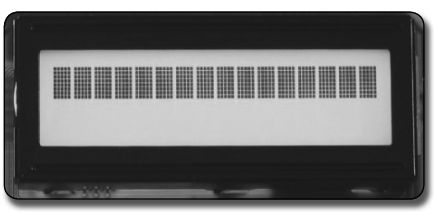
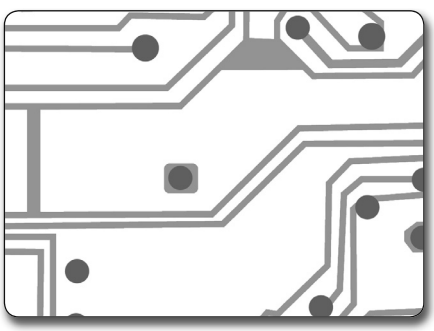


Figura 5. Quando ligado o LCD mostra esse aspecto



Listagem 8. Método para configurar os parâmetros de controle do display

```
procedure THD44780.Cursor_Display(CursorOn, DisplayOn, BlinkCursor : Boolean);
var
  Temp : byte;
begin
  Temp := LCD_ONOFF;
  if CursorOn then Temp := Temp or LCD_ONOFF_C;
  if DisplayOn then Temp := Temp or LCD_ONOFF_D;
  if BlinkCursor then Temp := Temp or LCD_ONOFF_CP;
  SendCommand(Temp);
end;
```

Listagem 9. Método para posicionamento do cursor

```
procedure THD44780.PosCursor(X, Y : Integer);
var
  offset : Integer;
begin
  offset := 0;
  if X > Columns then
    begin
    X := 1;
    inc(Y);
    end;
  if Y > Lines then
    Y := 1;
  if Y = 1 then offset := 0;
  if Y = 2 then offset := 64;

  SendCommand(LCD_DDRAM + offset + (X-1));
end;
```

Listagem 10. Método para deslocar cursor e/ou texto

```
procedure THD44780.ShiftCursor_Message(ShiftText, ToRight : Boolean);
var
  Temp : byte;
begin
  Temp := LCD_DESLOCA;
  if ShiftText then Temp := Temp or LCD_DESLOCA_M;
  if ToRight then Temp := Temp or LCD_DESLOCA_D;
  SendCommand(Temp);
end;
```

como dados e não como comandos) . Depois do *delay* enviamos os dados ao *display* e finalizamos baixando o *Enable* e aguardando o tempo de processamento do *LCD*.

Inicialização do LCD

Finalmente utilizando a rotina de envio de comandos *SendCommand*, inicializaremos o *display* com o método *InitializeLCD* como mostrado a seguir:

```
procedure THD44780.InitializeLCD;
begin
  SendCommand(LCD_FUNCAO or LCD_FUNCAO_B or
LCD_FUNCAO_L);
  sleep(15);
end;
```

No *InitializeLCD* configuramos o modo de operação enviando *LCD_FUNCAO* que será a máscara de bits para o modo de utilização do *LCD,* ou seja, diremos ao *LCD* que queremos configurar o modo de utilização, portanto dentro desta máscara de bits principal, estaremos agregando valores que irão configurar outros parâmetros. Usaremos a constante *LCD_FUNCAO_B,* que configura o modo de envio de dados para 8 bits e *LCD_FUNCAO_L* que configura o modo de 2 linhas para o *display*.

Repare que utilizamos o operador lógico *or* para que possamos configurar todos estes modos de operação ao mesmo tempo. E aguardamos um tempo de aproximadamente 15 milissegundos para completar a inicialização do *display*.

Parâmetros de controle do display

Com o *display* já inicializado, devemos configurar alguns parâmetros de controle de *display*, são eles: *cursor ativo, display ligado* e *cursor piscando*.

No método *Cursor_Display* teremos 3 parâmetros booleanos *CursorOn*, *DisplayOn*, *BlinkCursor* que utilizaremos para configurar o estado do display, conforme mostra a **Listagem 8**.

Nesta rotina, teremos a variável *Temp* que será o byte que desejamos enviar com a máscara de bits para o controle do *display* (usando a constante *LCD_ONOFF*), com isso faremos uma varredura para sabermos em quais parâmetros temos o valor *True* e mediante o uso do operador *or* agregamos o valor da respectiva constante a máscara do controle do *display LCD_ONOFF*. Depois desta varredura enviamos o valor como um comando ao *display* usando *SendCommand*.

Limpando o LCD

Para que possamos limpar o LCD sempre que desejarmos, precisaremos criar agora o comando de limpeza do *display*, bastando apenas enviar o valor da constante *LCD_LIMPA*, utilizando o método *SendCommand*, conforme listado a seguir. Crie uma nova *procedure* com o nome *ClearDisplay* e digite o código como mostrado. Não há segredo. Depois bastará fazer uma chamada a esse método em algum controle de nosso formulário:

```
procedure THD44780.ClearDisplay;
begin
  SendCommand(LCD_LIMPA);
end;
```

Posicionando o cursor

É possível controlar também o posicionamento do cursor, para isso necessitamos utilizar a constante *LCD_DDRAM* que nos proporciona acesso ao endereço da posição de dados do *display* (*DDRAM*) veja a **Listagem 9**.

Observe o leitor que através dos parâmetros X e Y do *PosCursor* podemos apontar uma nova posição do cursor, caso o valor de X seja superior ao número de colunas. Então colocamos a variável X na primeira posição da próxima linha. E caso a variável Y seja maior que o número de linhas atribuímos a Y o valor de 1, ou seja, posicionamos o cursor na primeira linha.

Repare nas duas últimas instruções *if*. Estamos tratando da primeira linha, então a variável *offset* recebe o valor de 0, do contrário recebe o valor 64, este *offset* se refere a posição da *DDRAM*.

Com isso podemos enviar o comando através do *SendCommand* utilizando a constante *LCD_DDRAM* para que o *LCD* entenda que se trata do endereçamento da posição do cursor e agregaremos o

valor do *offset* (que representa a linha em que queremos posicionar o cursor) e a posição da variável X – 1 (pois a primeira linha de endereços da *DDRAM* vai de 0 a 15).

Deslocando o cursor e/ou o texto

O *LCD HD44780* é muito versátil, podemos deslocar o cursor, o texto ou ambos ao mesmo tempo. Para isso utilizaremos a constante *LCD_DESLOCA* como máscara de bits principais para acessar a instrução para deslocar o cursor e texto. Para isso digite o conteúdo da **Listagem 10** no método *ShiftCursor_Message*.

Neste método receberemos 2 parâmetros do tipo *Boolean*, o primeiro *ShifText* que se for *True* faz com que além do cursor o texto também seja deslocado e o segundo *ToRight* que se for *True* desloca o texto e/ou cursor para direita, caso contrário faz o deslocamento para esquerda. Repare que adicionamos o valor das constantes a nossa variável local *Temp* através de operações com *or*. Enviando em seguida o valor acumulado em *Temp* com o comando *SendCommand*.

Escrevendo caracteres no LCD

Creio que agora chegamos ao método mais aguardado de todos, enviar texto ao *LCD*. Para realizar esta tarefa faremos uso do método *SendData* criado anteriormente, podemos ver a implementação do método *WriteText* na **Listagem 11**.

Note que para que possamos enviar o parâmetro *Text*, que é uma *string*, ao *LCD* fazemos uso de um laço *for* do primeiro caractere até o número máximo de caracteres recuperados através do método *Length* da variável *Text*, enviando um caractere por vez (tendo sido convertido para seu valor da tabela *ASCII* através função *Ord*). Toda vez que receber um caracter, o *LCD* o escreve na tela e move o cursor um espaço para a direita. O cursor marca o local onde o próximo caracter vai ser escrito.

Listagem 11. Método escrever caracteres no display

```
procedure THD44780.WriteText(Text : string);
var
  i : byte;
begin
  for i := 1 to Length(Text) do
    SendData(Ord(Text[i]));
end;
```

Listagem 12. Clique do btnInicializar

```
procedure TfrmMain.btnInicializarClick(Sender: TObject);
begin
  LCD.InitializeLCD;
  LCD.ClearDisplay;
  sleep(1);
  LCD.WriteText('Controle de LCD');
  LCD.PosCursor(1,2);
  LCD.WriteText('em Delphi!');
  LCD.Cursor_Display(True,True,True);
end;
```

Agora que já temos nossa classe pronta, podemos voltar à construção de nossa aplicação exemplo, declarando na seção *uses* da *uMain* a *unit uHD44780* que contém a classe e os métodos de controle do *LCD*. Não se esqueça de declarar uma variável do tipo *THD44780* conforme segue:

```
LCD : THD44780;
```

Com essa variável teremos acesso a todas as funcionalidades já implementadas de nossa classe especial, e poderemos controlar o *LCD* sem maiores dificuldades. Agora dê um duplo clique no *frmMain* e no evento *OnCreate* digite o código a seguir:

```
LPTPort1.Port[$378];
LCD := THD44780.Create(LPTPort1);
```

Com este código associamos o componente *LPTPort1* que colocamos em nosso *form* principal ao endereço $378 da porta paralela que é endereço base da porta, e instanciamos a variável *LCD* do tipo *THD44780*. Passamos como parâmetro do método *Create* o *LPTPort1* do nosso *form*, associando assim o *LPTPort1* a nossa variável de acesso ao *display*.

Não se esqueça de no evento *Destroy* do *frmMain* liberar o *LCD* da memória com o seguinte comando:

```
LCD.Free;
```

Precisamos agora de um *Button* para abrir a lista das portas paralelas do micro e um *ComboBox* onde armazenaremos a lista de portas. Coloque um *Button* com a propriedade *Name* alterada para *btnAbrir* e um *ComboBox* renomeado para *cmbxLPT*. Dê um duplo clique no *btnAbrir* e digite o código que segue no evento *OnClick* do botão:

```
LCD.LPTPort.Open;
```

Utilizando o método *Open* do *LPTPort* carregamos todas as *dlls* necessárias para acesso ao *hardware*, a partir daqui já temos permissão para listar as portas do micro. Para isso no evento *DropDown* do *cmbxLPT* coloque a seguinte linha de código:

```
LCD.LPTPort.EnumPorts(cmbxLPT.Items);
```

Com esta linha de código listamos todas as portas paralelas existentes no computador e armazenamos esta lista no *cmbxLPT*. Já temos agora todo controle sobre as portas paralelas e partiremos ao controle e inicialização do *LCD*. Coloque 3 *Buttons* cujas propriedades *Name* serão alteradas respectivamente para *btnInicializar*, *btnEscrever* e *btnLimpar*. No evento *OnClick* do *btnInicializar* iremos inicializar o *LCD* através do código da **Listagem 12.**

No clique do *btnInicializar*, faremos uso do método *InitializeLCD* que fará a inicialização do *display*, e só para nos certificarmos enviamos também um comando para limpar o *display* e fazemos um *delay* de 1 milissegundo. Depois

Nota do DevMan

O LCD HD44780 16x2 possui 32 posições para caracteres, ou seja, 16 posições para cada linha. O DDRAM ou Display Data RAM é uma memória volátil usada para armazenar os caracteres do display, possuindo 80 bytes, destes 80 apenas 32 posições são visíveis (as que são mostradas no display), caso escrevamos algum caractere fora destas 32 posições ele fica armazenado na DDRAM.

Listagem 13. Código de deslocamento de cursor ou texto (SpeedButtons)

```
procedure TfrmMain.spdbtnCursorEsqClick(Sender: TObject);
begin
  LCD.ShiftCursor_Message(False, False);
end;

procedure TfrmMain.spdbtnCursorDirClick(Sender: TObject);
begin
  LCD.ShiftCursor_Message(False, True);
end;

procedure TfrmMain.spdbtnTextoEsqClick(Sender: TObject);
begin
  LCD.ShiftCursor_Message(True, False);
end;

procedure TfrmMain.spdbtnTextoDirClick(Sender: TObject);
begin
  LCD.ShiftCursor_Message(True, True);
end;
```

disto com o método *WriteText* escrevemos o texto *Controle de LCD*, lembrando que assim que inicializado, o cursor do *LCD* já está apontando para a primeira posição da linha 1.

Colocamos o cursor na primeira posição da segunda linha usando o método *PosCursor* e nesta posição escrevemos na segunda linha o texto *em Delphi!* e ativamos o cursor, o *display* e cursor piscando com o método *Cursor_Display* passando todos os parâmetros como *True*, veja a tela do *LCD* após o clique do *btnInicializar* na **Figura 6**.

Voltando aos demais botões, digite no evento *OnClick* do *btnLimpar* o comando de limpeza do *display*:

```
LCD.ClearDisplay;
```

Para escrever texto através de nosso exemplo necessitamos de um *Edit*. Coloque um componente *Edit* no *form* principal e altere seu nome para *edtMensagem*, todo texto que digitarmos nele será enviado através do clique do *btnEscrever* que segue:

```
LCD.WriteText(edtMensagem.Text);
```

Agora colocaremos 3 *CheckBoxes* que utilizaremos para alterar a forma como o cursor está sendo mostrado. Depois de colocá-los, altere seus nomes para *chckbxCursorLigado*, *chckbxDisplayLigado* e *chckbxCursorPiscando*, respectivamente. Queremos que toda vez que alterar a propriedade *Checked* de qualquer um deste *CheckBoxes* o *LCD* atualize as configurações do cursor, para isso dê um duplo clique no *chckbxCursorLigado* e digite o código a seguir:

```
LCD.Cursor_
  Display(chckbxCursorLigado.Checked,
  chckbxDisplayLigado.Checked,
  chckbxCursorPiscando.Checked);
```

Aponte os eventos dos outros *CheckBoxes* para o evento *chckbxCursorLigadoClick*.

Movendo o cursor e/ou a mensagem

Para finalizar nosso exemplo necessitamos de 4 *SpeedButtons* (*spdbtnCursorEsq*, *spdbtnCursorDir*, *spdbtnTextoEsq*, *spdbtnTextoDir*) que serão utilizados para deslocar o cursor e a mensagem para a direita ou esquerda, utilizando o método *ShiftCursor_Message*. A **Listagem 13** apresenta o código dos *SpeedButtons*.

Repare que apenas alteramos os parâmetros do método *ShiftCursor_Message* que são utilizados para deslocar texto e cursor. Agora já temos nossa aplicação pronta para demonstrar algumas funcionalidades de controle de *LCD*. Na **Figura 7** temos uma sugestão de GUI que o caro leitor pode utilizar durante o desenvolvimento do programa.

Conclusão

Com nossa classe de controle de *LCDs* fica muito fácil interfacear e controlar *displays*, que são uma grande alternativa aos *hobbistas* e praticantes de *CaseModding*. É importante estudarmos essas metodologias que nos ajudam no desenvolvimento de diversas soluções. Recomendo a leitura de outros artigos e referências a esse tema. Fico por aqui, um forte abraço, e até a próxima. ●

Figura 6. Tela do LCD após clique do botão de inicialização

Figura 7. Sugestão de GUI para a aplicação exemplo

Links

Understanding User and Kernel Mode
http://www.codinghorror.com/blog/archives/001029.html

NT Drivers Faq
http://www.cmkrnl.com/faq01.html#1.2

LCD Interfacing
http://www.8051projects.net/lcd-interfacing/

Accessing Port Hardware
http://sheepdogguides.com/dt4j.htm

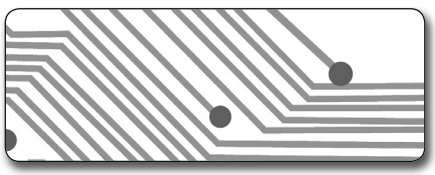
HD44780
http://pic.rocklizard.org/ReferenceDocs/HD44780HowTo.pdf

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



VENHA FAZER PARTE DA MELHOR EQUIPE DE HOSTING DO BRASIL

MAIS DE 100 VAGAS PARA RJ E SP

Acesse o site **www.alog.com.br** e cadastre seu currículo no link **carreira**

- SUPORTE TÉCNICO
- MONITORAMENTO
- GERÊNCIA DE SERVIÇOS
- OPERAÇÕES
- REDES
- INFRA-ESTRUTURA

ALOG DATA CENTERS DO BRASIL
Líder em Hosting Gerenciado

- Dois datacenters próprios - RJ e SP
- 99% de renovação contratual
- 850 clientes corporativos
- 8 mil m² de área construída
- Melhor empresa de internet pelo INFO 200 de 2008
- Destaque do ano no segmento Serviços de Internet do Anuário Telecom 2007

www.alog.com.br I 0800 282 3330

ALOG
DATA CENTERS DO BRASIL

Hosting Gerenciado® I Colocation I Contingência I Email Corporativo I Conectividade I Serviços Profissionais

Nesta seção você encontra artigos para iniciantes na linguagem Delphi

Introdução à paleta Servers

Integre sua aplicação com Word e Excel

Resumo DevMan

Os componentes da paleta Server são componentes muito poderosos para integrar suas aplicações do Delphi à documentos do Office como planilhas do Excel, documentos do Word, envio de e-mails com Outlook, apresentações do Power Point entre outras funcionalidades.

Nesse artigo veremos

- Utilização de componentes para integração do Delphi com Excel e Word;
- Exportação de registros do grid para Excel;
- Simulação de preenchimento de campos de um contrato do Word com Delphi.

Qual a finalidade

- Entender como usar e facilitar integração entre os programas.

Quais situações utilizam esses recursos?

- Em aplicações de cadastros com preenchimento de formulários para contratos, rotinas de exportação de relatórios.

A integração entre aplicativos em muitos dos casos acaba sendo o grande diferencial de um sistema, levando em consideração a capacidade do sistema interagir e trabalhar em outro *software* sem que haja qualquer intervenção por parte de usuários. Observamos muito estes aspectos quando há a digitação de resultados em um aplicativo comercial, onde as informações ficam armazenadas em uma base de dados, chaveadas para que a única forma possível de manipular seja através deste sistema. Supondo para este caso que um usuário necessite de algum relatório, alguma planilha específica e que a mesma não esteja contemplada no sistema, uma solução fácil seria disponibilizar a ele uma lista de *fields* disponíveis para serem trabalhadas e possibilitar que as informações sejam exportadas e salvas em uma planilha do *Excel*, podendo ser aplicados filtros, criação de tabelas

Maikel Marcelo Scheid
maikelscheid@gmail.com
é técnico em Informática com ênfase em Análise e Programação de Sistemas. Atua na área de Desenvolvimento de Softwares em Delphi para plataforma Win32 e .NET com banco de dados Firebird e MS SQL. É membro da equipe Editorial ClubeDelphi.

dinâmicas, gráficos e outras formas de visualizar as informações.

Baseado em situações como esta ou situações como, por exemplo, a inclusão de um contrato de locação a um sistema comercial e a necessidade de atualização e impressão de um documento físico do *Word* contendo cláusulas e informações diversas do contrato, onde uma integração entre sistemas poderia fazer este trabalho de forma automatizada, otimizando o tempo necessário para o procedimento, veremos neste artigo que para ambas as situações isto será possível e não muito complicado de ser produzido.

Utilizaremos para o desenvolvimento de exemplos do artigo o *Delphi 7* e o componente *TClientDataSet* para acesso a dados através de arquivos *XML*. Faremos utilização de componentes da paleta *Servers* e veremos duas alternativas de exportação das informações do *ClientDataSet* para o *Excel*, sendo uma delas onde o sistema escreve individualmente o valor em cada linha/coluna do *Excel* e outra onde faremos uma exportação em memória, exibindo as informações exportadas apenas no momento da conclusão.

Em uma terceira situação, trabalharemos com um documento do *Word* que indica o cadastro de um novo contrato no sistema, fazendo após a atualização do documento com informações preenchidas.

Criando a aplicação

No menu *File|New>Application* crie um novo projeto *Win32* e salve a *unit* e o projeto em um diretório como *uServers.pas* e *prjServers.dpr* respectivamente. Altere a propriedade *name* do formulário principal para *FrmServers* e o *Caption* para *Componentes da Paleta Servers*. Faremos a utilização de um componente *PageControl* (*pgGuias*) da paleta *Win32* para que possamos dividir o sistema em duas partes, uma na qual iremos exemplificar a exportação dos registros de um *ClientDataSet* para o *Excel* e na outra a manipulação de informação contidas em documentos do *Word*. No componente *pgGuias*, altere a propriedade *Align* para *alClient* fazendo com que utilize todo o espaço do *Form* e com o botão direito crie duas novas páginas (*New Page*) alterando a propriedade *Name* da *TabSheet1* para *TSExcel* e *Caption* para *Paleta Servers - Excel* e na *TabSheet2* altere o *Name* para *TSWord* e o *Caption* para *Paleta Servers - Word*.

Figura 1. Layout do sistema para integração de aplicativos

Na *TSExcel*, criaremos um processo para que informações sejam carregadas de um arquivo *XML* utilizando um componente *ClientDataSet* e exibí-las em um componente *DBGrid*, que será utilizado para realizar a exportação para as planilhas do *Excel*. Adicione um componente *Button* (*btnCarregarXML*) que utilizaremos para abrir um diálogo de busca do *XML*, adicione também da paleta *Data Controls* um componente *DBGrid* (*gridXML*).

Para carregarmos em memória as informações de um arquivo *XML*, precisamos adicionar da paleta *Dialogs* um componente *OpenDialog* (*dlgXML*) para exibir o diálogo de busca e da paleta *Data Access* os componentes *ClientDataSet* (*CDSLoadXML*) que carregará o *XML* e um *DataSource* (*DSLoadXML*) que irá relacionar o *CDSLoadXML* ao *gridXML* exibindo os dados carregados. No *DSLoadXML* ligue a propriedade *DataSet* ao *CDSLoadXML* e em seguida selecione o *gridXML* relacionando-o através de sua propriedade *DataSource* ao componente *DSLoadXML* concluindo esta etapa de relacionamento dos componentes.

Para a exportação das informações a uma planilha do *Excel*, veremos o processo de duas formas distintas; uma delas onde o sistema abrirá uma nova planilha (visível ao usuário) e fará a transcrição das informações contidas no *DataSet* linha a linha, preenchendo individualmente cada célula da planilha aberta até que chegue ao final dos registros e outro processo onde o sistema irá percorrer todos os registros, criando uma tabela em memória e somente ao final do último irá transcrever os dados para a planilha do Excel, exibindo-o ao usuário.

Para estas exportações adicione logo abaixo ao *gridXML* dois *Buttons* (*btnExportaExcelTela* e *btnExportaExcelMem*) alterando o *Caption* dos mesmos para *Exportar Grid para Excel - na tela* e para *Exportar Grid para Excel - em memória*. Adicione também da paleta *Servers* os componentes principais da exportação, dois componentes *ExcelApplication* (*ExcelApp* e *ExcelAppMem*) para a exportação dos dados na tela e em memória, também precisaremos adicionar os componentes *ExcelWorkbook* (*ExcelWbkMem*) e *ExcelWorksheet* (*ExcelWstMem*) a serem utilizados na exportação em memória a ser realizada. Adicionados todos os componentes visuais e não visuais à aplicação (**Figura 1**), daremos início ao processo de codificação desta primeira etapa de integração dos aplicativos.

A primeira codificação a ser realizada será o processo para a pesquisa e carregamento do arquivo *XML* onde iremos

Listagem 1. Codificando a pesquisa do arquivo XML

```
procedure TfrmServers.btnCarregaXMLClick(Sender: TObject);
begin
  dlgXML.Filter := 'Arquivos XML (*.xml)|*.xml|Todos os Arquivos |*.*';
  { Define o filtro do OpenDialog para mostrar apenas extensões *.xml }
  dlgXML.Execute;
  { Abre a caixa de diálogo para pesquisa dos arquivos }
  if ExtractFileExt(dlgXML.FileName) = '.xml' then
  { Verifica se o arquivo selecionado é XML }
  begin
    { Carrega no ClientDataSet o XML selecionado }
    CDSLoadXML.Close;
    CDSLoadXML.LoadFromFile(dlgXML.FileName);
  end
  else
  MessageDlg('Arquivo selecionado não é XML válido!',mtError,[mbOK],0);
  { Mostra mensagem de erro se arquivo diferente de XML }
end;
```

Listagem 2. Exportando dados para Excel

```
procedure TfrmServers.btnExportaExcelTelaClick(Sender: TObject);
var
   Planilha   : olevariant;
   Lin, Col   : integer;
begin
  { Cria a aplicação do excel }
  ExcelApp := TExcelApplication.Create(nil);
  { Adiciona novo documento }
  ExcelApp.Workbooks.Add(EmptyParam,0);
  { Permite visualizar a exportação em tempo real}
  ExcelApp.Visible[0] := True;
  { Utiliza o primeiro documento em branco do documento excel }
  Planilha := ExcelApp.Workbooks[1].Worksheets[1];
  { Cria a linha de cabeçalho da aplicação }
  for Col := 1 to CDSLoadXML.FieldCount do
  begin
    Planilha.Cells[1,Col].Select;
    Planilha.Cells[1,Col].Font.Bold := True;
    Planilha.Cells[1,Col].Value := CDSLoadXML.Fields[Col-1].DisplayLabel;
  end;
  { Inicia a transcrição dos registros na linha 2 }
  Lin := 2;
  { Enquanto houver registros no laço, vai escrevendo os mesmos no Excel }
  while not ( CDSLoadXML.Eof ) do
  begin
    { Adiciona os registros de acordo com cada coluna do excel }
    for Col := 1 to CDSLoadXML.FieldCount do
    begin
      Planilha.Cells[Lin,Col].Select;
      Planilha.Cells[Lin,Col].Value := CDSLoadXML.Fields[Col-1].AsString;
    end;
    { Incrementa + 1 para avançar linha }
    inc(Lin);
    { Avança mais um registro no laço de contagem }
    CDSLoadXML.Next;
  end;
  { Fecha o componente de exibição dos dados }
  CDSLoadXML.Close;
  { Limpa a referência do ClientDataset }
   CDSLoadXML.Destroy ;
  { Libera a aplicação da memória }
  ExcelApp.Free;
end;
```

chamar o método *execute* do componente *OpenDialog* e ao selecionar um arquivo verificamos sua extensão. O arquivo a ser carregado deverá obrigatoriamente possuir a extensão **.xml* que irá fechar o *ClientDataSet* e carregar as informações do novo arquivo selecionado, que serão exibidas no componente *gridXML*. Neste processo, adicione ao evento *OnClick* do *btnCarregaXML* o código da **Listagem 1** que encontra-se também com comentários para melhor entendimento do andamento dos processos.

Realizada a codificação execute o sistema e faça uma busca de arquivos *XML* que poderão ser encontrados normalmente no diretório *C:\Arquivos de Programas\Arquivos Comuns\Borland Shared\Data* fazendo com que qualquer um dos arquivos sejam carregados para o sistema.

Integração com o Excel

Já realizada e testada a etapa de carregar os arquivos *XML*, faremos agora a codificação dos dois processos para a exportação das informações carregadas para o *Excel*. Na primeira codificação faremos a exportação por meio do processo onde as informações são diretamente transcritas, onde no evento *OnClick* do botão *btnExportaExcelTela* o código da **Listagem 2** deverá ser adicionado. O código encontra-se também comentado de acordo com as principais funcionalidades. Durante o processo de execução deste código, será criada uma nova aplicação do *Excel* e a ela adicionados uma planilha ainda sem dados, exibindo-a ao usuário. Ao trabalhar e acessar o *ClientDataSet* fazemos um *loop* por todos os *fields* adicionando na primeira linha os nomes de cada coluna

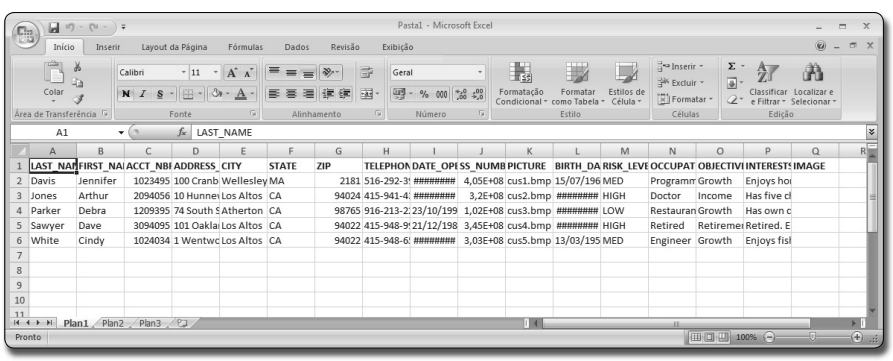


Figura 2. Tabela clients.xml exportada para o Excel

Nota do DevMan

Os componentes da paleta Servers na verdade são apenas wrappers (componentes que encapsulam) interfaces de bibliotecas COM. Em outras palavras, é apenas uma forma mais simples de você interagir com aplicativos OLE (como o Word e Excel) sem conhecer detalhes específicos, como o uso de Type Libraries (bibliotecas de tipos).

do *CDSLoadXML* criando a identificação no *Excel*. No segundo laço, posicionamos o cursor no primeiro registro carregado e logo após vamos percorrendo os mesmos, adicionando-os um a um no *Excel*, onde todo o processo está sendo acompanhado pelo usuário. Ao inserir o último registro, fechamos e destruímos o espaço utilizado pelo *CDSLoadXML* na memória e finalizamos a aplicação *Excel* criada, não nos preocupando mais em salvar a planilha que está visível ao usuário e este mesmo poderá trabalhar com os dados e salvar o documento se preferir.

Depois de realizado o processo de codificação a primeira etapa do sistema já poderá ser executada e testada, carregando um arquivo de *XML* e logo após realizar a exportação dos dados para o *Excel* (**Figura 2**).

O processo para a codificação do segundo processo de exportação é bastante semelhante ao primeiro, alterando, porém a forma com que as informações serão transcritas ao *Excel*. Neste processo precisaremos adicionar ao evento *OnClick* do botão o código da **Listagem 3** que inicialmente irá trabalhar com os registros do *CDSLoadXML* recolhendo as informações

Listagem 3. Exportando dados para Excel utilizando tabela em memória

```
procedure TfrmServers.btnExportaExcelMemClick(Sender: TObject);
var
  { Variáveis para ser utilizadas na exportação }
  y :integer;
  tsList :TStringList;
  s :string;
  aSheet :Variant;
  Clipboard1 : TClipboard;
begin
  try
    { Posiciona o CDSLoadXML no primeiro registro }
    CDSLoadXML.First;
    { Cria um novo objeto Stringlist }
    tsList:=TStringList.Create;
    s := '';
    { Percorre a quantidade de colunas do CDSLoadXML para adicionar a primeira linha ao Excel }
    for y:=0 to CDSLoadXML.FieldCount-1 do
    begin
      { Adiciona nomes as colunas de acordo com colunas do CDSLoadXML }
      s:=s+CDSLoadXML.Fields[y].DisplayLabel+#9;
      Application.ProcessMessages;
    end;
    { Adiciona a lista obtida ao StringList }
    tsList.Add(s);
    { Percorre todos os registros carregados no CDSLoadXMLDS }
    while not CDSLoadXML.Eof do
    begin
      s := '';
      { Percore as colunas do registro e carrega informações da linha }
      for y:=0 to CDSLoadXML.FieldCount-1 do
      begin
        { Verifica se coluna não é Blob - imagem }
        if not CDSLoadXML.Fields[y].IsBlob then
        { Adiciona registro da coluna na linha para exportação }
        s:=s+CDSLoadXML.Fields[y].AsString+#9;
        { Processa as ações do sistema }
        Application.ProcessMessages;
      end;
      { Adiciona a tabela de registros ao StringList }
      tsList.Add(s);
      { Avança um registro }
      CDSLoadXML.next;
    end;
  finally
    { Fecha e remove CDSLoadXMLDS da memória }
    CDSLoadXML.Close;
    CDSLoadXML.Destroy ;
  end;
  try
    { Adiciona a tabela do StringList na memória }
    if Clipboard1 = nil then
    begin
      { Cria a área de transferência }
      Clipboard1 := TClipboard.create;
    end;
    { Joga a tabela para a área de transferência }
    Clipboard1.AsText:=tsList.Text;
  finally
    { Remove objeto de memória }
    Clipboard1.Destroy ;
    Clipboard1 := nil ;
  end;
  try
    try
      { Conecta ao Excel }
      ExcelAppMem.connect;
    except
      exit;
    end;
    { Mostra o Excel }
    ExcelAppMem.Visible[0]:=True;
    { Descrição do trabalho no Excel }
    ExcelAppMem.Caption:='Exportação ao Excel';
    { Adiciona nova planilha em branco }
    ExcelAppMem.Workbooks.Add(Null,0);
    { Conecta à planilha criada }
    ExcelWbkMem.ConnectTo(ExcelAppMem.Workbooks[1]);
    ExcelWstMem.ConnectTo(ExcelWbkMem.Worksheets[1] as _Worksheet);
    { Cola a tabela em memória para a planilha criada }
    ExcelWstMem.Paste ;
  finally
    { Desconecta do Excel }
    ExcelAppMem.Disconnect;
  end;
end;
```

Listagem 4. Código do evento OnClick do botão btnContrato

```
procedure TfrmServers.btnContratoClick(Sender: TObject);
var
  DocOrigem , DocDestino : OleVariant;
  Procurar  , Substituir : OleVariant;
  i : Integer;
begin
  { Abre o diálogo para pesquisa do Modelo }
  dlgWord.Execute;
  if (ExtractFileExt(dlgWord.FileName) = '.dot') then
  begin
    { Cria uma nova aplicação no Word }
    WordApp:=TWordApplication.Create(Application.Owner);
    { Faz com que o Word seja aberto e visível ao usuário }
    WordApp.Visible:=True;
    { Adiciona um novo documento ao Word }
    WordDoc:=TWordDocument.Create(Application.Owner);
    { Local do documento de Origem }
    DocOrigem := dlgWord.FileName;
    { Abre o documento de Origem no Word }
    WordDoc.ConnectTo(WordApp.Documents.Add(
      DocOrigem,EmptyParam,EmptyParam,EmptyParam));
    for i := 0 to ComponentCount -1 do
    begin
      if Components[i] is TEdit then
      begin
        { Pesquisa pela chave com nome do componente, igual ao que criamos no Word }
        Procurar := '@'+Components[i].Name;
        { Substitui a chave pelo texto do componente }
        Substituir := (Components[i] as TEdit).Text;
        { Localiza e substitui as strings no Documento aberto }
        WordDoc.Content.Find.Execute(
          Procurar,EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
          EmptyParam,EmptyParam,Substituir,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
          EmptyParam);
      end;
    end;
    (* Define caminho e nome para salvar novo arquivo *)
    DocDestino := 'C:\Contrato.DOC';
    (* Salva *)
    WordDoc.SaveAs(
      DocDestino,EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,EmptyParam,
      EmptyParam,EmptyParam,EmptyParam);
    { Fecha o documento }
    WordDoc.Close;
  end
  else
  { Mostra mensagem de formato incorreto }
  MessageDlg('Arquivo selecionado não está no formato correto.',mtWarning,[mbOK],0);
end;
```

do título das colunas e adicionando-as como cabeçalho da primeira linha a ser exportada para o *Excel*. Numa segunda etapa, trabalhamos diretamente com os registros armazenados no componente, percorrendo-os de um a um, recolhendo informações e adicionando-as ao objeto *StringList*, que após passar pelo último registro do *CDSLoadXML* estará carregado com todas suas informações. Terminadas as atividades com o *CDSLoadXML*, criamos uma nova área de transferência que ira receber a estrutura de tabela que está armazenada no *StringList*. Somente neste seguinte passo, trabalharemos propriamente dito com o *Excel*, criando uma nova aplicação, criando uma nova planilha de informações e posteriormente colando as informações armazenadas na área de transferência para a nova planilha do *Excel*, que somente neste momento será exibida ao usuário.

Finalizada mais esta etapa de códigos, faça a execução do sistema e carregue novamente um arquivo de dados *XML* para o *CDSLoadXML* e desta vez utilize-se do botão *Exportar Grid para Excel - em memória* para realizar a exportação das informações, onde a planilha lhe será exibida apenas ao término de todos os registros percorridos.

Finalizamos com esta funcionalidade a demonstração de integração do *Delphi*

com *Excel*, onde você poderá escolher uma das formas apresentadas e realizar a implementação desta funcionalidade em seus sistemas. Veremos nesta próxima funcionalidade a integração do *Delphi* com documentos do *Word*.

Integrando o Delphi ao Microsoft Word

Para este exemplo prático criaremos uma estrutura que simula o cadastro de um contrato de locação, adicionando informações da parte contratada e objeto a ser locado. Para esta estrutura, posicione-se na guia *Paleta Servers - Word* (*TSWord*) onde precisaremos adicionar alguns componentes para digitação. Arraste para o formulário 11 componentes *Label* alterando a propriedade *Caption* de todos eles seguindo a ordem a seguir: *NÚM. CONTRATO*, *DATA*, *CLIENTE*, *ENDEREÇO*, *CIDADE*, *UF*, *CPF*, *IE*, *LOCAÇÃO DO EQUIPAMENTO*, *FRANQUIA* e *VALOR*. Os componentes deverão ser organizados de forma que possa ser adicionado um componente *Edit* ao lado de cada um deles, sendo que outros 11 componentes *Edit* (*edtNumero*, *edtData*, *edtCliente*, *edtEndereco*, *edtCidade*, *edtUF*, *edtCPF*, *edtIE*, *edtEquipamento*, *edtFranquia*, *edtValor*) deverão também ser arrastados e organizados. Arraste ainda logo abaixo de todos os componentes adicionados um componente *Button* (*btnContrato*) que será o responsável por integrar a aplicação ao *Word*. Precisamos adicionar ainda neste formulário um novo componente *OpenDialog* (*dlgWord*) para localizarmos o arquivo de modelo do contrato e adicionar também os componentes principais para realizar a integração, arrastando da paleta *Servers* os componentes *WordDocument* (*WordDoc*) que será utilizado para abrir o documento de modelo e trabalhar com o texto salvo e um componente *WordApplication* (*WordApp*) que propriamente dito será a integração com o aplicativo do *Word*. Ao final da organização dos componentes no formulário, o *layout* do formulário (**Figura 3**) estará pronto para ser codificado e integrado com o *Word*, onde precisamos criar nosso documento do contrato modelo (**Figura 4**).

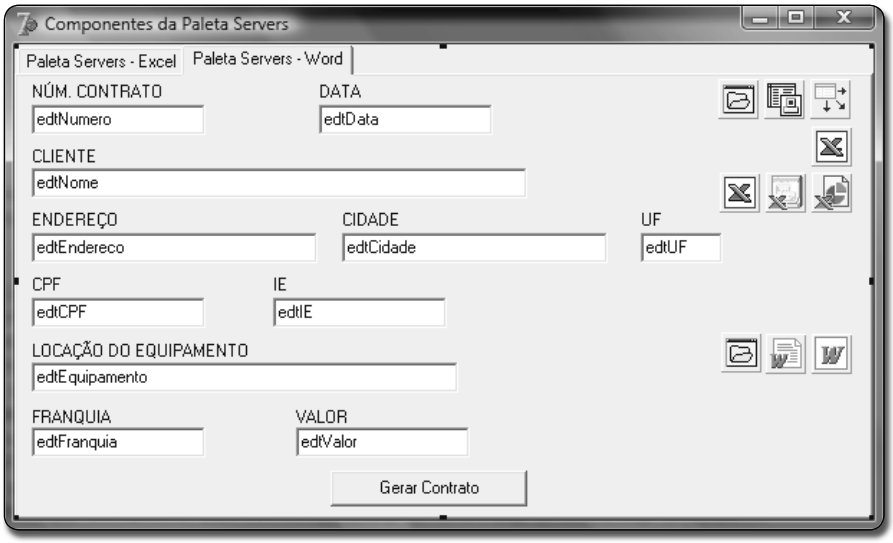


Figura 3. Estrutura para integração do Delphi ao Word

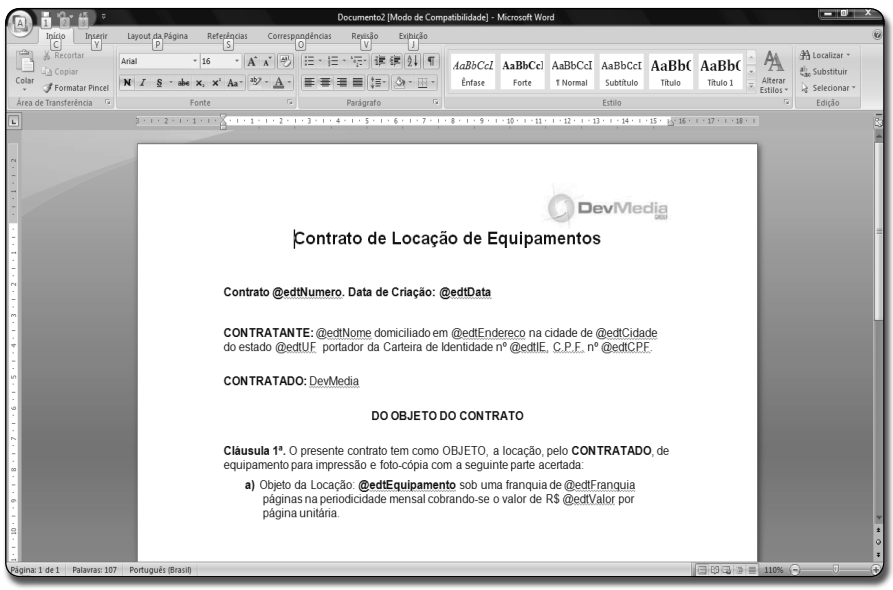


Figura 4. Modelo padrão do contrato

No *Word*, precisamos criar um contrato que será nosso objeto padrão para todos os contratos a serem criados e nele definir alguns campos que serão as chaves de identificação para substituição dos valores, ou seja, definimos que um dos campos chaves de identificação será o conjunto *@edtNumero*, que iremos depois referenciar no *Delphi* e proporcionar a substituição, da mesma forma como declaramos também as chaves *@edtData*, *@edtNome* e outras chaves mais. Note que utilizamos para as chaves do *Word* o nome dos campos que criamos no *Delphi* para os componentes a serem exportados. Salve o documento do *Word* com um formato **.dot* em um diretório padrão e deverá ser localizado em *runtime*.

Retornando a nossa aplicação, atribua ao evento *OnClick* do botão *btnContrato* o código da **Listagem 4** onde criamos um novo aplicativo do *Word* visível ao usuário e adicionamos a ele um documento, no qual será carregado o arquivo de modelo selecionado através da caixa de diálogo da busca. Para substituição dos trechos, utilizamos um processo adaptado perfeitamente para este exemplo, onde temos campos do tipo *TEdit* somente para cadastro de informações do contrato e definimos

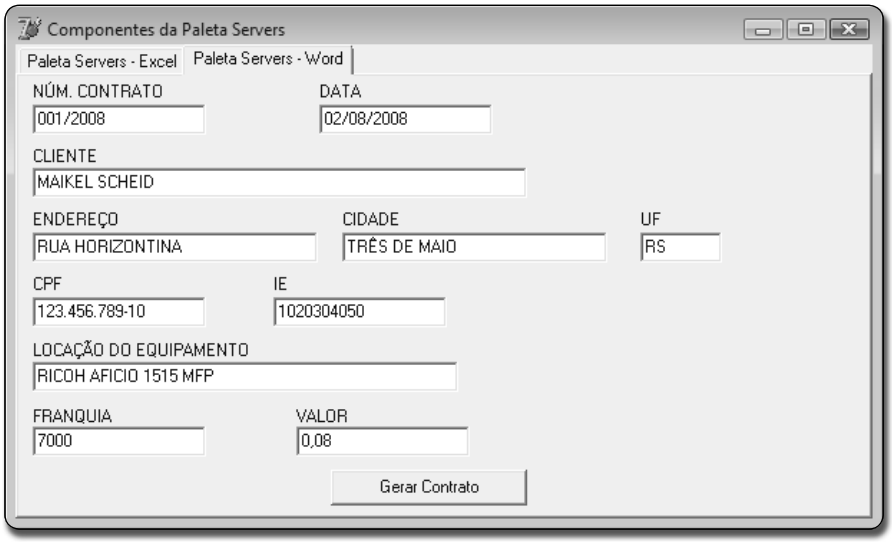


Figura 5. Tela preenchida com informações do contrato

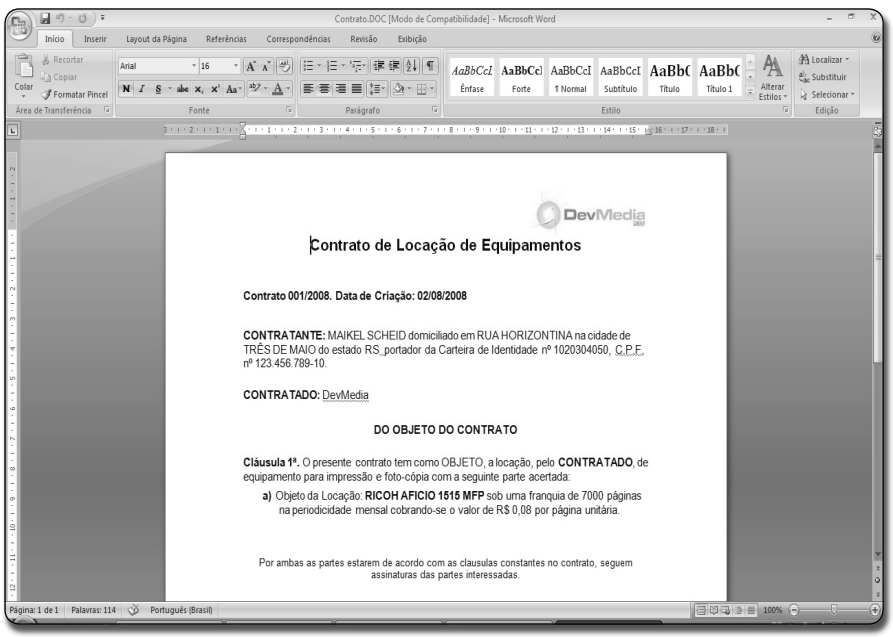


Figura 6. Contrato com informações preenchidas de acordo com aplicativo

no modelo do *Word* os mesmos nomes das chaves que temos para o campos no formulário. Neste processo verificamos todos os componentes, e ao encontrar um objeto do tipo *TEdit* localizamos a chave no documento e ao encontrá-la substituímos pelo valor do componente. Ao finalizar todo o processo de substituição, definimos um nome para o novo documento e salvamos o mesmo em um diretório do computador.

Após inserido todo o código ao evento *OnClick* e criado o modelo de documento do *Word*, poderemos realizar o teste de nossa aplicação, onde ao executá-la preencha os campos com alguns valores (**Figura 5**) e clique para *Gerar Contrato* onde verás uma aplicação do *Word* sendo aberta e fechada na tela e logo após verifique no local de destino a criação de um novo documento com as informações dos campos chaves substituídos pelos valores definidos no aplicativo (**Figura 6**).

Conclusão

Vimos nestes exemplos algumas formas de integrar aplicativos *Delphi* a documentos no *Word* e *Excel*, onde exportamos valores de objetos *TClientDataSet* e também substituímos textos em documentos do *Word*. Veja como estes processos podem ser utilizados em seus aplicativos e crie outras formas de trabalhar com estas informações, exporte imagens e crie gráficos para estes documentos, utilize também outros recursos disponibilizados por meio dos demais componentes da paleta *Servers*. Abraço e até o próximo artigo. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Existem coisas que não conseguimos ficar sem!

...só pra lembrar, sua assinatura pode estar acabando!

Renove Já!

www.devmedia.com.br/renovacao

DevMedia GROUP

Para mais informações:
www.devmedia.com.br/central

Nesta seção você encontra artigos sobre a linguagem PHP e a ferramenta Delphi for PHP

Trabalhando com classes no PHP

Veja como implementar e utilizar na prática classes em Web Sites PHP

Adriano Santos
(falecom@adrianosantos.pro.br)
é desenvolvedor Delphi desde 1998. Professor e programador PHP. Bacharel em Comunicação Social pela Universidade Cruzeiro do Sul, SP. É Editor Técnico, Colunista e Membro da Comissão Editorial das revistas ClubeDelphi e WebMobile. Gerente de Desenvolvimento SoftPark, empresa parceira Borland. Mantém o blog Delphi to Delphi (www.delphitodelphi.blogspot.com) com dicas, informações e tudo sobre desenvolvimento Delphi. Um dos fundadores do grupo Delphi Users Group de São Paulo - DUG-SP (www.dug-sp.com).

Resumo DevMan

A palavra-chave é: reaproveitamento. Reaproveitar código agiliza a vida dos desenvolvedores envolvidos no projeto. Com classes, é possível reaproveitar código e dessa forma centralizar o desenvolvimento do sistema. Veremos nesse artigo como criar e utilizar classes no PHP.

Nesse artigo veremos

Conceito de classes no PHP;
Qual a utilidade das classes;
Por que e como utilizar classes;

Qual a finalidade

Entender como funcionam e como utilizar classes no dia-a-dia;

Quais situações utilizam esses recursos?

Aplica-se a todo e qualquer Web Site desenvolvido em PHP. É um dos princípios da orientação a objetos;

Acredito que o sucesso de qualquer empresa que se proponha a desenvolver sistemas, seja Web ou Desktop, está no planejamento da aplicação e do negócio em si. Um sistema bem planejado com certeza terá vida longa e evidentemente dará bons frutos. Na engenharia de software, podemos enumerar diversos aspectos que uma aplicação precisa ter para ser tornar "uma aplicação de sucesso". Nem todas as empresas vivem apenas de um software, ou um aplicativo. Muitas software-houses comercializam mais de um sistema ao mesmo tempo e muitas vezes para ramos de atividades diversos, o que torna o negócio ainda mais arriscado.

Quando falamos em sistemas para Web, o risco é ainda maior, pois muitas vezes um projeto nada tem a ver com outro. Nesse caso o desenvolvimento acaba sendo individualizado tornando a aplicação única. Independente disso,

muitas vezes nos deparamos com situações corriqueiras e que precisam ser aplicadas diversas vezes em aplicações diferentes, por isso acabamos criando *functions* e *procedures* para que possamos reaproveitar código e o tempo gasto, claro.

Em aplicações *Win32*, com o bom e velho Delphi, isso é moleza. Podemos simplesmente criar um componente, uma *Unit* repleta de funções, uma *Dll* ou mesmo uma classe que o problema está resolvido, ou seja, temos mais formas de solução do problema do que imaginamos. Mas e na Web? O que podemos fazer para evitar a repetição de código e ainda por cima garantir uma bom sucesso para nosso sistema?

Em PHP a resposta para isso são as classes. Assim como em Delphi podemos criar "units" que servem como bibliotecas de funções. Para incorporá-las aos nossos Web Sites precisamos invocar as funções *include()* ou *include_once()*. Feito isso, as funções, variáveis, classes, constantes, *arrays*, etc ali presentes passam a ser vistas pela página que invocou o método. Em outras palavras, nosso problema de reaproveitamento de código está solucionado. Vamos à prática saber como funcionam as classes em PHP e como utilizá-las. Mão na massa!

Listagem 1. Código de consulta a clientes

```
01 <?
02   $conexao = mysql_connect("localhost", "root", "masterkey");
03   mysql_select_db("clubedelphi");
04   $sql = "select * from clientes";
05   $query = mysql_query($sql);
06 ?>
07
08 <table width="700" border="1">
09  <tr>
09     <td><b>Nome</b></td>
10     <td><b>Endereço</b></td>
11     <td><b>Cidade</b></td>
12     <td><b>UF</b></td>
13     <td><b>CEP</b></td>
14   </tr>
15 <?
16   while ($row = mysql_fetch_array($query)) {
17 ?>
18   <tr>
19     <td><? print $row[0]; ?></td>
20     <td><? print $row[1]; ?></td>
21     <td><? print $row[2]; ?></td>
22     <td><? print $row[3]; ?></td>
23     <td><? print $row[4]; ?></td>
24   </tr>
25 <?
26 }
27   mysql_free_result($query);
28   mysql_close($conexao);
29 ?>
30 </table>
```

Criando o primeiro exemplo

Classes nada mais são do que uma coleção de funções agrupadas em um arquivo que pode ser comparado, grosseiramente, a uma *Unit* do *Delphi for Win32*. Podemos criar diversas funções para as mais diversas tarefas em um ou mais arquivos. A grande vantagem nisso, certamente, é a reutilização de código já que não precisamos reprogramar tarefas quando já as temos prontas em algum lugar. Acho que as duas tarefas mais importantes e certamente que mais se repetem em uma aplicação são a conexão e desconexão com o banco de dados. Por isso vamos utilizar esse exemplo para entendermos bem como e quando utilizar classes.

Vamos escrever uma aplicação simples utilizando o banco de dados *MySQL 6.0.6 Alpha* que pode ser encontrado em versão de demonstração no *www.superdownloads.com.br*. Não faremos a instalação bem como a criação do banco de dados, por não fazer parte do escopo desse artigo. Apenas tenha em mente uma simples a base com duas tabelas como Clientes e Funcionários. Os campos podem ser apenas os necessários, tais como Nome, Endereço, Cidades, Estado e CEP. A estrutura pode ser a mesma, inclusive, para ambas as tabelas.

Para o primeiro script, que fará a listagem dos clientes em nosso banco de dados, abra o Bloco de Notas do Windows ou qualquer editor (no meu caso estou utilizando uma versão mais antiga do *Dreamweaver* da *Macromedia* que já me proporciona maior comodidade) de sua preferência e digite o código da **Listagem 1**.

O código da **Listagem 1** é bem simples de se entender. Entre as linhas 1 e 6 perceba que abrimos e fechamos *tags* *<?* e *?>*, o que significa que o código ali escrito é totalmente código PHP. Primeiro criamos uma variável que é associada ao método *mysql_connect()* que tem por objetivo conectar-se ao banco de dados. Passamos para o método o *host*, onde encontra-se o servidor do *MySQL* rodando, nesse caso *localhost*, o usuário (*root*) e a senha. Na linha logo abaixo selecionamos qual o banco de dados desejamos usar para essa conexão e por fim executamos uma consulta usando como parâmetro a instrução *SQL* recebida pela variável *$sql*. Entre as linhas 8 e 14 criamos uma tabela *HTML* usando a *tag* *<table>* mais uma linha (*<tr>*) com algumas células (*<td>*) que servirão de cabeçalho, de título para as informações que vamos gerar.

Logo em seguida, nas linhas 15 a 17, abrimos e fechamos novamente uma seção *PHP* para efetuar um loop na consulta (*$query*). Perceba que fechamos a *tag PHP* entramos novamente com *HTML* na linha 18 até 24. Nesse ponto estamos mesclando PHP com HTML. Repare no trecho a seguir:

```
19 <td><? print $row[0]; ?></td>
```

Aqui eu peguei uma linha proposital para mostrar que estamos usando PHP e HTML numa só linha de programação. Esse é um macete bastante utilizado por programadores *PHP*. Abrimos uma *tag* de célula (*<td>*), codificamos chamando o método *print()* para impressão de um campo da tabela Clientes (*<? **print** $row[0]; ?>*) e em seguida fechamos a *tag* célula com *</td>*. Repetimos isso para as demais linhas modificando apenas o índice do campo na variável *$row*, que fará menção a cada campo da tabela como se fosse o *FieldByName()* ou *Fields[X]* no *Delphi for Win32*.

Listagem 2. Código de consulta a funcionários

```
01 <?
02   $conexao = mysql_connect("localhost", "root", "masterkey");
03   mysql_select_db("clubedelphi");
04   $sql = "select * from funcionarios";
05   $query = mysql_query($sql);
06 ?>
07
08 <table width="700" border="1">
09  <tr>
09     <td><b>Nome</b></td>
10     <td><b>Endereço</b></td>
11     <td><b>Cidade</b></td>
12     <td><b>UF</b></td>
13     <td><b>CEP</b></td>
14     <td><b>E-Mail</b></td>

15   </tr>
16 <?
17   while ($row = mysql_fetch_array($query)) {
18 ?>
19   <tr>
20     <td><? print $row[0]; ?></td>
21     <td><? print $row[1]; ?></td>
22     <td><? print $row[2]; ?></td>
23     <td><? print $row[3]; ?></td>
24     <td><? print $row[4]; ?></td>
25     <td><? print $row[5]; ?></td>
26   </tr>
27 <?
28 }
29   mysql_free_result($query);
30   mysql_close($conexao);
31 ?>
32 </table>
```

Figura 1. Lista de clientes cadastrados

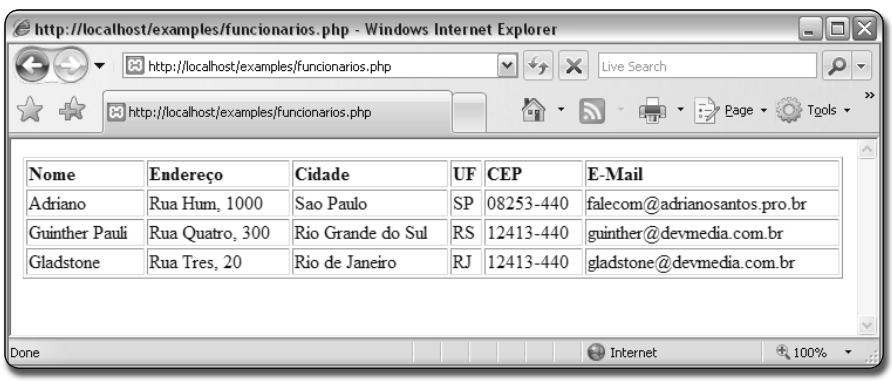


Figura 2. Lista de funcionários cadastrados

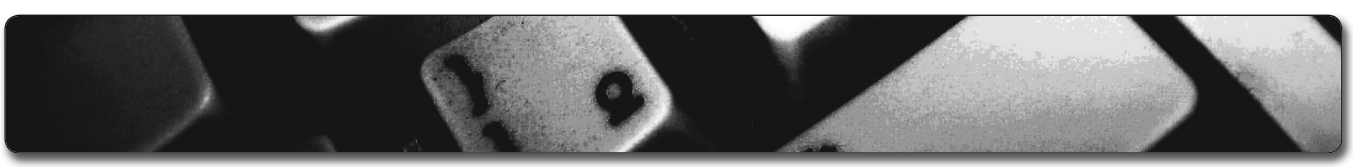
No finalzinho do *script* limpamos o conteúdo da variável *$query* e fechamos a conexão. O resultado desse script pode ser visto na **Figura 1**. Salve esse arquivo como *clientes.php*, execute e veja o resultado.

Muito bom, agora vamos imaginar que precisamos da mesma pesquisa, porém na tabela de Funcionários. Como fazer? Fácil, crie um novo arquivo no Bloco de Notas ou editor de sua preferência e então copie e cole o script do primeiro arquivo. Nessa tabela, funcionários, a única diferença é que temos um campo a mais: email, ou seja, deveremos incorporar esse campo em mais uma coluna de nossa janela de consulta. Veja como o *script* fica depois da alteração na **Listagem 2**. Em seguida rode a aplicação e veja o resultado (**Figura 2**).

Vejamos o que alteramos no *script* para que se adapte a tabela de funcionários. Repare nas linhas 04, 14 e 25. Na linha 04 encontra-se nossa instrução SQL, onde precisamos modificar o nome da tabela de *clientes* para *funcionarios*. As linhas 14 e 25 precisaram ser incluídas, pois aumentamos uma coluna em nossa grade. A linha 14 contém o cabeçalho da coluna *E-Mail* e a 25 o conteúdo do campo de mesmo nome. Conseqüentemente nosso *script* agora contém *32* linhas, *2* a mais que o anterior. Mas a quantidade de linhas não quer dizer nada, quero apenas que se foque em entender nosso problema.

O trecho entre as linhas 01 e 06, 29 e 31 não sofreram grandes alterações. Porém, uma coisa me preocupa: o serviço de hospedagem do nosso cliente foi cancelado e precisamos remover o site desse servidor e migrar para outro. O *host* onde o banco de dados encontra-se alocado, no novo servidor, tem uma *url* um pouco diferente. A invés de usarmos *localhost*, teremos que usar:

```
localhost:3306/path/empresa/
```



Veja o transtorno. Teremos que acessar todas as páginas que possuem conexão com o banco de dados e modificar a linha:

```
$conexao = mysql_connect("
  localhost", "root", "masterkey");
```

Para:

```
$conexao = mysql_connect("
  localhost:3306/path/empresa/","
  root", "masterkey");
```

Nossa aplicação possui apenas dois arquivos, mas imagine um Web Site legado com mais de trezentas páginas e diversos processos de conexão e desconexão em cada uma delas!? Com uma boa documentação e paciência, seria bem fácil efetuar a migração, porém torça para que seu cliente não cancele tão cedo o contrato com esse novo serviço de hospedagem.

Esse é só um dos inúmeros exemplos que posso citar, onde temos que parar e pensar em uma solução. Sim, podemos criar uma "unit" com funções de conexão e desconexão do banco, recebendo como parâmetro o endereço do banco, a *url*. Mas por que não criarmos uma classe para que possamos usar em todo o sistema e que poupe linhas de programação?

Entendendo e criando as classes

A primeira coisa que faremos é criar um novo *script PHP* para receber nossas classes, portanto crie um novo arquivo e salve-o como *classes.php*. Salve no mesmo diretório dos arquivos anteriores. Inicie a criação da classe digitando o trecho seguinte:

```
<?
   ## Essa é nossa primeira classe
   class TMySQL{
   }
?>
```

Perceba que, propositalmente criei a classe com o prefixo "T" na frente, como em Delphi, mas não é necessário. Poderia colocar qualquer nome na classe desde que não se inicie por caracteres especiais ou números. Apenas uma ressalva: lembre-se de que o *PHP* é sensitivo, ou seja, *TMySQL* é diferente de *TMYSQL* ou *Tmysql* etc. Portanto, você deve lembrar-se como foi declarada a classe, para que possa usá-la corretamente, ou então, crie padrões próprios.

O segundo passo é criar variáveis globais que serão usadas ao logo de todo o código da classe e fazer sua implementação. Para tal, criaremos uma função dentro da classe que será responsável por conectar-se ao banco de dados e emitir uma resposta dizendo se conseguiu ou não conectar-se. Implemente o código de nossa classe como na **Listagem 3**.

Repare que declaramos cinco variáveis globais da classe nas linhas de 04 a 08 e logo em seguida declaramos outras variáveis locais dentro da *function conectar*. A *function* recebe como parâmetro as variáveis responsáveis pelo endereço do servidor *MySQL*, banco de dados a ser utilizado, usuário e senha, respectivamente. Em seguida a função tenta conectar-se ao banco informado e retorna se obteve sucesso ou não.

Ainda em nossa *classes.php* inclua uma nova *function* que fará executar uma instrução SQL no banco, veja como é sua implementação na **Listagem 4**.

A explicação é simples: recebemos a instrução SQL no parâmetro da função e tentamos executá-la. Caso haja sucesso ou rejeição, enviamos uma mensagem ao usuário.

Uma última providência antes de fazer o teste de uso da classe. Crie um novo arquivo chamado *globais.ini* e coloque-o no diretório de sua aplicação. Nele declare algumas variáveis como segue:

```
<?
  $servidor = "localhost";
  $banco_dados = "clubedelphi";
  $usuario = "root";
  $senha = "masterkey";
?>
```

Listagem 3. Código de consulta a funcionários

```
<?
## Essa é nossa primeira classe
class TMySQL{
  var $servidor;
  var $base_de_dados;
  var $usuario;
  var $senha;
  var $socket_conexao;
  function conectar($host, $db, $user, $pass) {
    $this->servidor = $host;
    $this->base_de_dados = $db;
    $this->usuario = $user;
    $this->senha = $pass;
    $this->socket_conexao=@mysql_connect($this->servidor,$this->
       usuario,$this->senha);
    if (!$this->socket_conexao){
      return false;
    }else{
      if (!@mysql_select_db($this->base_de_dados,$this->socket_conexao)){
        return false;
      }else{
        return true;
      }
    }
  }
 }
?>
```

Listagem 4. Código de consulta dentro da classe

```
class TMySQL{
...
  ## Declare essa variável no início da classe
  var $minha_consulta;
...
  ## Função de consulta
  function consultar($string){
    $this->minha_consulta = mysql_query($string,$this->socket);
    if (!$this->minha_consulta) {
      print "Erro manipulando query!";
    }else{
      if (substr($string,0,6)=="select") {
        print "Comando aceito!";
      }else{
        print "Isso não é um select";
      }
    }
  }
```

Listagem 5. Utilização na prática de Classes

```
<?
  include_once("classes.php");
  include("globais.ini");
  $MySQL = new TMySQL();
  if ($MySQL->conectar($servidor, $banco_dados, $usuario, $senha)){
    $MySQL->consultar("select * from clientes");
  }else{
    print "Não foi possível conectar-se ao banco de dados.";
  };
?>
```

Veja que declaramos em um arquivo separado dados importantes de conexão, dessa forma podemos usar a qualquer momento e numa eventual mudança, basta alterar o arquivo e reenviar ao site que toda a aplicação estará modificada.

Agora para entender como usar a classe, crie um novo arquivo com o nome que desejar e digite o código da **Listagem 5**. Perceba que usamos dois métodos para importar os arquivos *classes.php* e *globais.ini*. Desse modo o script atual conseguirá visualizar os dados, funções, variáveis, classes, etc de cada arquivo. Em seguida criamos a classe, chamamos o método *conectar* e em seguida *consultar*. Poderíamos fazer um exemplo ainda mais completo, com funções de desconexão, geração automática da grade, etc, mas isso prefiro deixar a título de estudos.

Conclusão

Nesse artigo vimos a real necessidade em se utilizar classes. E vimos também que não é difícil. As classes nada mais são que funções agrupadas em um arquivo único, como um componente ou DLL. Procure estudar mais o desenvolvimento de classes. Esse é um dos princípios da orientação a objetos. Boa sorte e até a próxima. ●

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback

Conhecimento faz diferença!

Mais de 60 mil downloads na primeira edição!

Faça já sua assinatura digital! | *www.devmedia.com.br/es*

Faça um *up grade* em sua carreira.

Em um mercado cada vez mais focado em qualidade ter conhecimentos aprofundados sobre requisitos, metodologia, análises, testes, entre outros, pode ser a diferença entre conquistar ou não uma boa posição profissional. Sabendo disso a DevMedia lança para os desenvolvedores brasileiros sua primeira revista digital totalmente especializada em Engenharia de Software. Todos os meses você irá encontrar artigos sobre Metodologias Ageis; Metodologias tradicionais (*document driven*); ALM (*application lifecycle management*); SOA (aplicações orientadas a servicos); Analise de sistemas; modelagem; Métricas; orientação à objetos; UML; testes e muito mais. **Assine já!**

DevMedia GROUP

A INVASÃO DE PROPRIEDADE NÃO SE LIMITA A RESIDÊNCIAS.

Assegure-se que seu software está protegido pelo líder em segurança da informação, antes que seja tarde demais.

SafeNet tem à sua disposição a proteção de software mais avançada e segura. As Chaves de Hardware Sentinel oferecem tecnologia que incorpora criptografia de chave pública, criptografia AES e autenticação interna para garantir o mais alto nível de proteção anti-pirataria.

Peça seu Kit de Desenvolvimento hoje mesmo.

Entre em contato em +55 11 4208-7700 em SP-Brasil
ou por e-mail infopirataria@safenet-inc.com

25 YEARS
1983–2008

SafeNet